VISIT…

INTRODUCTION

TO

MACHINE LEARNING

AN EARLY DRAFT OF A PROPOSED

TEXTBOOK

Nils J. Nilsson
Robotics Laboratory
Department of Computer Science
Stanford University
Stanford, CA 94305

e-mail: nilsson@cs.stanford.edu

December 4, 1996

Copyright ©1997 Nils J. Nilsson
This material may not be copied, reproduced, or distributed without the written permission of the copyright holder.

Contents

1 Preliminaries **1**

1.1 Introduction 1

1.1.1 What is Machine Learning? 1

1.1.2 Wellsprings of Machine Learning 3

1.1.3 Varieties of Machine Learning 5

1.2 Learning Input-Output Functions 6

1.2.1 Types of Learning 6

1.2.2 Input Vectors 8

1.2.3 Outputs 9

1.2.4 Training Regimes 9

1.2.5 Noise 10

1.2.6 Performance Evaluation 10

1.3 Learning Requires Bias 10

1.4 Sample Applications 13

1.5 Sources 14

1.6 Bibliographical and Historical Remarks 15

2 Boolean Functions **17**

2.1 Representation 17

2.1.1 Boolean Algebra 17

2.1.2 Diagrammatic Representations 18

2.2 Classes of Boolean Functions 19

2.2.1 Terms and Clauses 19

2.2.2 DNF Functions 20

2.2.3 CNF Functions 24
2.2.4 Decision Lists 25
2.2.5 Symmetric and Voting Functions 26
2.2.6 Linearly Separable Functions 26
2.3 Summary 27
2.4 Bibliographical and Historical Remarks 28

3 Using Version Spaces for Learning 29
3.1 Version Spaces and Mistake Bounds 29
3.2 Version Graphs 31
3.3 Learning as Search of a Version Space 34
3.4 The Candidate Elimination Method 35
3.5 Bibliographical and Historical Remarks 37

4 Neural Networks 39
4.1 Threshold Logic Units 39
4.1.1 Definitions and Geometry 39
4.1.2 Special Cases of Linearly Separable Functions 41
4.1.3 Error-Correction Training of a TLU 42
4.1.4 Weight Space 45
4.1.5 The Widrow-Hoff Procedure 46
4.1.6 Training a TLU on Non-Linearly-Separable Training Sets 49
4.2 Linear Machines 50
4.3 Networks of TLUs 51
4.3.1 Motivation and Examples 51
4.3.2 Madalines 54
4.3.3 Piecewise Linear Machines 56
4.3.4 Cascade Networks 57
4.4 Training Feedforward Networks by Backpropagation 58
4.4.1 Notation 58
4.4.2 The Backpropagation Method 60
4.4.3 Computing Weight Changes in the Final Layer . . . 62
4.4.4 Computing Changes to the Weights in Intermediate Layers 64

4.4.5 Variations on Backprop 66
4.4.6 An Application: Steering a Van 67
4.5 Synergies Between Neural Network and Knowledge-Based Methods 68
4.6 Bibliographical and Historical Remarks 68

5 Statistical Learning **69**
5.1 Using Statistical Decision Theory 69
5.1.1 Background and General Method 69
5.1.2 Gaussian (or Normal) Distributions 71
5.1.3 Conditionally Independent Binary Components . . . 75
5.2 Learning Belief Networks 77
5.3 Nearest-Neighbor Methods 77
5.4 Bibliographical and Historical Remarks 79

6 Decision Trees **81**
6.1 Definitions 81
6.2 Supervised Learning of Univariate Decision Trees 83
6.2.1 Selecting the Type of Test 83
6.2.2 Using Uncertainty Reduction to Select Tests 84
6.2.3 Non-Binary Attributes 88
6.3 Networks Equivalent to Decision Trees 88
6.4 Overfitting and Evaluation 89
6.4.1 Overfitting 89
6.4.2 Validation Methods 90
6.4.3 Avoiding Overfitting in Decision Trees 91
6.4.4 Minimum-Description Length Methods 92
6.4.5 Noise in Data 93
6.5 The Problem of Replicated Subtrees 94
6.6 The Problem of Missing Attributes 96
6.7 Comparisons 96
6.8 Bibliographical and Historical Remarks 96

7 Inductive Logic Programming **97**

7.1 Notation and Definitions 99
7.2 A Generic ILP Algorithm 100
7.3 An Example 103
7.4 Inducing Recursive Programs 107
7.5 Choosing Literals to Add 110
7.6 Relationships Between ILP and Decision Tree Induction . . 111
7.7 Bibliographical and Historical Remarks 114

8 Computational Learning Theory **117**

8.1 Notation and Assumptions for PAC Learning Theory 117
8.2 PAC Learning 119
8.2.1 The Fundamental Theorem 119
8.2.2 Examples 121
8.2.3 Some Properly PAC-Learnable Classes 122
8.3 The Vapnik-Chervonenkis Dimension 124
8.3.1 Linear Dichotomies 124
8.3.2 Capacity 126
8.3.3 A More General Capacity Result 127
8.3.4 Some Facts and Speculations About the VC Dimension 129
8.4 VC Dimension and PAC Learning 129
8.5 Bibliographical and Historical Remarks 130

9 Unsupervised Learning **131**

9.1 What is Unsupervised Learning? 131
9.2 Clustering Methods 133
9.2.1 A Method Based on Euclidean Distance 133
9.2.2 A Method Based on Probabilities 136
9.3 Hierarchical Clustering Methods 138
9.3.1 A Method Based on Euclidean Distance 138
9.3.2 A Method Based on Probabilities 138
9.4 Bibliographical and Historical Remarks 143

10 Temporal-Difference Learning **145**
10.1 Temporal Patterns and Prediction Problems 145
10.2 Supervised and Temporal-Difference Methods 146
10.3 Incremental Computation of the $(\Delta \mathbf{W})_i$ 148
10.4 An Experiment with TD Methods 150
10.5 Theoretical Results 152
10.6 Intra-Sequence Weight Updating 153
10.7 An Example Application: TD-gammon 155
10.8 Bibliographical and Historical Remarks 156

11 Delayed-Reinforcement Learning **159**
11.1 The General Problem 159
11.2 An Example 160
11.3 Temporal Discounting and Optimal Policies 161
11.4 Q-Learning 164
11.5 Discussion, Limitations, and Extensions of Q-Learning . . . 167
11.5.1 An Illustrative Example 167
11.5.2 Using Random Actions 169
11.5.3 Generalizing Over Inputs 170
11.5.4 Partially Observable States 171
11.5.5 Scaling Problems 172
11.6 Bibliographical and Historical Remarks 173

12 Explanation-Based Learning **175**
12.1 Deductive Learning 175
12.2 Domain Theories 176
12.3 An Example 178
12.4 Evaluable Predicates 182
12.5 More General Proofs 183
12.6 Utility of EBL 183
12.7 Applications 183
12.7.1 Macro-Operators in Planning 184
12.7.2 Learning Search Control Knowledge 186
12.8 Bibliographical and Historical Remarks 187

Preface

These notes are in the process of becoming a textbook. The process is quite unfinished, and the author solicits corrections, criticisms, and suggestions from students and other readers. Although I have tried to eliminate errors, some undoubtedly remain—*caveat lector*. Many typographical infelicities will no doubt persist until the final version. More material has yet to be added. Please let me have your suggestions about topics that are too important to be left out. I hope that future versions will cover Hopfield nets, Elman nets and other recurrent nets, radial basis functions, grammar and automata learning, genetic algorithms, and Bayes networks .... I am also collecting exercises and project suggestions which will appear in future versions.

Some of my plans for additions and other reminders are mentioned in marginal notes.

My intention is to pursue a middle ground between a theoretical textbook and one that focusses on applications. The book concentrates on the important *ideas* in machine learning. I do not give proofs of many of the theorems that I state, but I do give plausibility arguments and citations to formal proofs. And, I do not treat many matters that would be of practical importance in applications; the book is not a handbook of machine learning practice. Instead, my goal is to give the reader sufficient preparation to make the extensive literature on machine learning accessible.

Students in my Stanford courses on machine learning have already made several useful suggestions, as have my colleague, Pat Langley, and my teaching assistants, Ron Kohavi, Karl Pfleger, Robert Allen, and Lise Getoor.

Chapter 1

Preliminaries

1.1 Introduction

1.1.1 What is Machine Learning?

Learning, like intelligence, covers such a broad range of processes that it is difficult to define precisely. A dictionary definition includes phrases such as "to gain knowledge, or understanding of, or skill in, by study, instruction, or experience," and "modification of a behavioral tendency by experience." Zoologists and psychologists study learning in animals and humans. In this book we focus on learning in machines. There are several parallels between animal and machine learning. Certainly, many techniques in machine learning derive from the efforts of psychologists to make more precise their theories of animal and human learning through computational models. It seems likely also that the concepts and techniques being explored by researchers in machine learning may illuminate certain aspects of biological learning.

As regards machines, we might say, very broadly, that a machine learns whenever it changes its structure, program, or data (based on its inputs or in response to external information) in such a manner that its expected future performance improves. Some of these changes, such as the addition of a record to a data base, fall comfortably within the province of other disciplines and are not necessarily better understood for being called learning. But, for example, when the performance of a speech-recognition machine improves after hearing several samples of a person's speech, we feel quite justified in that case to say that the machine has learned.

Machine learning usually refers to the changes in systems that perform tasks associated with *artificial intelligence (AI)*. Such tasks involve recognition, diagnosis, planning, robot control, prediction, etc. The "changes" might be either enhancements to already performing systems or *ab initio* synthesis of new systems. To be slightly more specific, we show the architecture of a typical AI "agent" in Fig. 1.1. This agent perceives and models its environment and computes appropriate actions, perhaps by anticipating their effects. Changes made to any of the components shown in the figure might count as learning. Different learning mechanisms might be employed depending on which subsystem is being changed. We will study several different learning methods in this book.

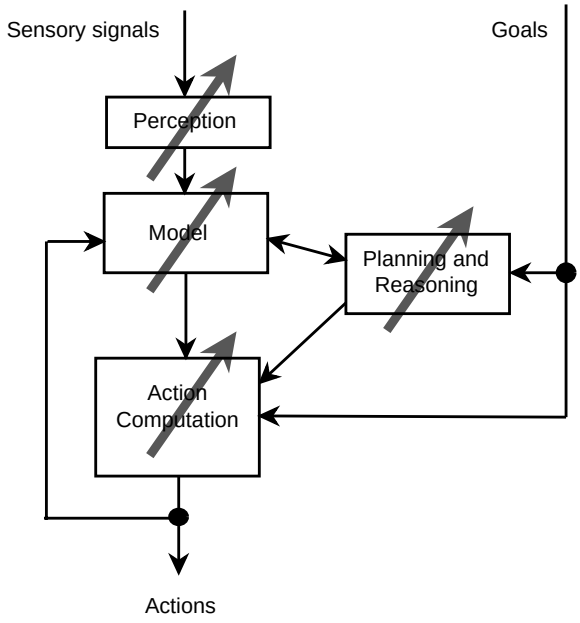


Figure 1.1: An AI System

One might ask "Why should machines have to learn? Why not design machines to perform as desired in the first place?" There are several reasons why machine learning is important. Of course, we have already mentioned that the achievement of learning in machines might help us understand how animals and humans learn. But there are important engineering reasons as well. Some of these are:

- Some tasks cannot be defined well except by example; that is, we might be able to specify input/output pairs but not a concise relationship between inputs and desired outputs. We would like machines to be able to adjust their internal structure to produce correct outputs for a large number of sample inputs and thus suitably constrain their input/output function to approximate the relationship implicit in the examples.

- It is possible that hidden among large piles of data are important relationships and correlations. Machine learning methods can often be used to extract these relationships (*data mining*).

- Human designers often produce machines that do not work as well as desired in the environments in which they are used. In fact, certain characteristics of the working environment might not be completely known at design time. Machine learning methods can be used for on-the-job improvement of existing machine designs.

- The amount of knowledge available about certain tasks might be too large for explicit encoding by humans. Machines that learn this knowledge gradually might be able to capture more of it than humans would want to write down.

- Environments change over time. Machines that can adapt to a changing environment would reduce the need for constant redesign.

- New knowledge about tasks is constantly being discovered by humans. Vocabulary changes. There is a constant stream of new events in the world. Continuing redesign of AI systems to conform to new knowledge is impractical, but machine learning methods might be able to track much of it.

1.1.2 Wellsprings of Machine Learning

Work in machine learning is now converging from several sources. These different traditions each bring different methods and different vocabulary which are now being assimilated into a more unified discipline. Here is a brief listing of some of the separate disciplines that have contributed to machine learning; more details will follow in the the appropriate chapters:

- **Statistics:** A long-standing problem in statistics is how best to use samples drawn from unknown probability distributions to help decide from which distribution some new sample is drawn. A related problem

is how to estimate the value of an unknown function at a new point given the values of this function at a set of sample points. Statistical methods for dealing with these problems can be considered instances of machine learning because the decision and estimation rules depend on a corpus of samples drawn from the problem environment. We will explore some of the statistical methods later in the book. Details about the statistical theory underlying these methods can be found in statistical textbooks such as [Anderson, 1958].

- **Brain Models:** Non-linear elements with weighted inputs have been suggested as simple models of biological neurons. Networks of these elements have been studied by several researchers including [McCulloch & Pitts, 1943, Hebb, 1949, Rosenblatt, 1958] and, more recently by [Gluck & Rumelhart, 1989, Sejnowski, Koch, & Churchland, 1988]. Brain modelers are interested in how closely these networks approximate the learning phenomena of living brains. We shall see that several important machine learning techniques are based on networks of nonlinear elements—often called *neural networks*. Work inspired by this school is sometimes called *connectionism*, *brain-style computation*, or *sub-symbolic processing*.

- **Adaptive Control Theory:** Control theorists study the problem of controlling a process having unknown parameters which must be estimated during operation. Often, the parameters change during operation, and the control process must track these changes. Some aspects of controlling a robot based on sensory inputs represent instances of this sort of problem. For an introduction see [Bollinger & Duffie, 1988].

- **Psychological Models:** Psychologists have studied the performance of humans in various learning tasks. An early example is the EPAM network for storing and retrieving one member of a pair of words when given another [Feigenbaum, 1961]. Related work led to a number of early decision tree [Hunt, Marin, & Stone, 1966] and semantic network [Anderson & Bower, 1973] methods. More recent work of this sort has been influenced by activities in artificial intelligence which we will be presenting.

  Some of the work in reinforcement learning can be traced to efforts to model how reward stimuli influence the learning of goal-seeking behavior in animals [Sutton & Barto, 1987]. Reinforcement learning is an important theme in machine learning research.

- **Artificial Intelligence:** From the beginning, AI research has been concerned with machine learning. Samuel developed a prominent early program that learned parameters of a function for evaluating board positions in the game of checkers [Samuel, 1959]. AI researchers have also explored the role of analogies in learning [Carbonell, 1983] and how future actions and decisions can be based on previous exemplary cases [Kolodner, 1993]. Recent work has been directed at discovering rules for expert systems using decision-tree methods [Quinlan, 1990] and inductive logic programming [Muggleton, 1991, Lavrač & Džeroski, 1994]. Another theme has been saving and generalizing the results of problem solving using explanation-based learning [DeJong & Mooney, 1986, Laird, *et al.*, 1986, Minton, 1988, Etzioni, 1993].

- **Evolutionary Models:**

  In nature, not only do individual animals learn to perform better, but species *evolve* to be better fit in their individual niches. Since the distinction between evolving and learning can be blurred in computer systems, techniques that model certain aspects of biological evolution have been proposed as learning methods to improve the performance of computer programs. Genetic algorithms [Holland, 1975] and genetic programming [Koza, 1992, Koza, 1994] are the most prominent computational techniques for evolution.

1.1.3 Varieties of Machine Learning

Orthogonal to the question of the historical source of any learning technique is the more important question of *what* is to be learned. In this book, we take it that the thing to be learned is a computational structure of some sort. We will consider a variety of different computational structures:

- Functions
- Logic programs and rule sets
- Finite-state machines
- Grammars
- Problem solving systems

We will present methods both for the synthesis of these structures from examples and for changing existing structures. In the latter case, the change

to the existing structure might be simply to make it more computationally efficient rather than to increase the coverage of the situations it can handle. Much of the terminology that we shall be using throughout the book is best introduced by discussing the problem of learning functions, and we turn to that matter first.

1.2 Learning Input-Output Functions

We use Fig. 1.2 to help define some of the terminology used in describing the problem of learning a function. Imagine that there is a function, f, and the task of the learner is to guess what it is. Our hypothesis about the function to be learned is denoted by h. Both f and h are functions of a vector-valued input $\mathbf{X} = (x_1, x_2, \ldots, x_i, \ldots, x_n)$ which has n components. We think of h as being implemented by a device that has $\mathbf{X}$ as input and $h(\mathbf{X})$ as output. Both f and h themselves may be vector-valued. We assume *a priori* that the hypothesized function, h, is selected from a class of functions $\mathcal{H}$. Sometimes we know that f also belongs to this class or to a subset of this class. We select h based on a *training set*, Ξ, of m input vector examples. Many important details depend on the nature of the assumptions made about all of these entities.

1.2.1 Types of Learning

There are two major settings in which we wish to learn a function. In one, called *supervised learning*, we know (sometimes only approximately) the values of f for the m samples in the training set, Ξ. We assume that if we can find a hypothesis, h, that closely agrees with f for the members of Ξ, then this hypothesis will be a good guess for f—especially if Ξ is large.

Curve-fitting is a simple example of supervised learning of a function. Suppose we are given the values of a two-dimensional function, f, at the four sample points shown by the solid circles in Fig. 1.3. We want to fit these four points with a function, h, drawn from the set, $\mathcal{H}$, of second-degree functions. We show there a two-dimensional parabolic surface above the x_1, x_2 plane that fits the points. This parabolic function, h, is our hypothesis about the function, f, that produced the four samples. In this case, $h = f$ at the four samples, but we need not have required exact matches.

In the other setting, termed *unsupervised learning*, we simply have a training set of vectors without function values for them. The problem in this case, typically, is to partition the training set into subsets, Ξ_1, ..., Ξ_R, in some appropriate way. (We can still regard the problem as one of

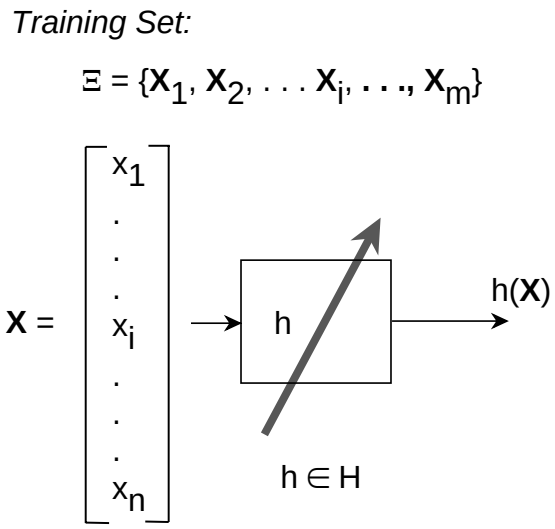


Figure 1.2: An Input-Output Function

learning a function; the value of the function is the name of the subset to which an input vector belongs.) Unsupervised learning methods have application in taxonomic problems in which it is desired to invent ways to classify data into meaningful categories.

We shall also describe methods that are intermediate between supervised and unsupervised learning.

We might either be trying to find a new function, h, or to modify an existing one. An interesting special case is that of changing an existing function into an equivalent one that is computationally more efficient. This type of learning is sometimes called *speed-up* learning. A very simple example of speed-up learning involves deduction processes. From the formulas $A \supset B$ and $B \supset C$, we can deduce C if we are given A. From this deductive process, we can create the formula $A \supset C$—a new formula but one that does not sanction any more conclusions than those that could be derived from the formulas that we previously had. But with this new formula we can derive C more quickly, given A, than we could have done before. We can contrast speed-up learning with methods that create genuinely new functions—ones that might give different results after learning than they did before. We say that the latter methods involve *inductive* learning. As opposed to deduction, there are no *correct* inductions—only useful ones.

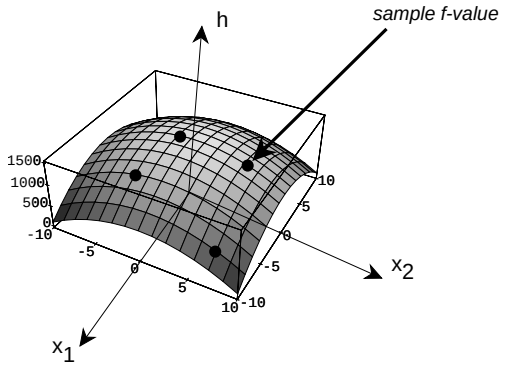


Figure 1.3: A Surface that Fits Four Points

1.2.2 Input Vectors

Because machine learning methods derive from so many different traditions, its terminology is rife with synonyms, and we will be using most of them in this book. For example, the input vector is called by a variety of names. Some of these are: *input vector*, *pattern vector*, *feature vector*, *sample*, *example*, and *instance*. The components, x_i, of the input vector are variously called *features*, *attributes*, *input variables*, and *components*.

The values of the components can be of three main types. They might be real-valued numbers, discrete-valued numbers, or *categorical values*. As an example illustrating categorical values, information about a student might be represented by the values of the attributes *class, major, sex, adviser*. A particular student would then be represented by a vector such as: (sophomore, history, male, higgins). Additionally, categorical values may be *ordered* (as in {*small, medium, large*}) or *unordered* (as in the example just given). Of course, mixtures of all these types of values are possible.

In all cases, it is possible to represent the input in unordered form by listing the names of the attributes together with their values. The vector form assumes that the attributes are ordered and given implicitly by a form. As an example of an *attribute-value* representation, we might have: (major: history, sex: male, class: sophomore, adviser: higgins, age: 19). We will be using the vector form exclusively.

An important specialization uses Boolean values, which can be regarded as a special case of either discrete numbers (1,0) or of categorical variables

(*True*, *False*).

1.2.3 Outputs

The output may be a real number, in which case the process embodying the function, h, is called a *function estimator*, and the output is called an *output value* or *estimate*.

Alternatively, the output may be a categorical value, in which case the process embodying h is variously called a *classifier*, a *recognizer*, or a *categorizer*, and the output itself is called a *label*, a *class*, a *category*, or a *decision*. Classifiers have application in a number of recognition problems, for example in the recognition of hand-printed characters. The input in that case is some suitable representation of the printed character, and the classifier maps this input into one of, say, 64 categories.

Vector-valued outputs are also possible with components being real numbers or categorical values.

An important special case is that of Boolean output values. In that case, a training pattern having value 1 is called a *positive instance*, and a training sample having value 0 is called a *negative instance*. When the input is also Boolean, the classifier implements a *Boolean function*. We study the Boolean case in some detail because it allows us to make important general points in a simplified setting. Learning a Boolean function is sometimes called *concept learning*, and the function is called a *concept*.

1.2.4 Training Regimes

There are several ways in which the training set, Ξ, can be used to produce a hypothesized function. In the *batch* method, the entire training set is available and used all at once to compute the function, h. A variation of this method uses the entire training set to modify a current hypothesis iteratively until an acceptable hypothesis is obtained. By contrast, in the *incremental* method, we select one member at a time from the training set and use this instance alone to modify a current hypothesis. Then another member of the training set is selected, and so on. The selection method can be random (with replacement) or it can cycle through the training set iteratively. If the entire training set becomes available one member at a time, then we might also use an incremental method—selecting and using training set members as they arrive. (Alternatively, at any stage all training set members so far available could be used in a "batch" process.) Using the training set members as they become available is called an *online* method.

Online methods might be used, for example, when the next training instance is some function of the current hypothesis and the previous instance—as it would be when a classifier is used to decide on a robot's next action given its current set of sensory inputs. The next set of sensory inputs will depend on which action was selected.

1.2.5 Noise

Sometimes the vectors in the training set are corrupted by noise. There are two kinds of noise. *Class noise* randomly alters the value of the function; *attribute noise* randomly alters the values of the components of the input vector. In either case, it would be inappropriate to insist that the hypothesized function agree precisely with the values of the samples in the training set.

1.2.6 Performance Evaluation

Even though there is no correct answer in inductive learning, it is important to have methods to evaluate the result of learning. We will discuss this matter in more detail later, but, briefly, in supervised learning the induced function is usually evaluated on a separate set of inputs and function values for them called the *testing set* . A hypothesized function is said to *generalize* when it guesses well on the testing set. Both mean-squared-error and the total number of errors are common measures.

1.3 Learning Requires Bias

Long before now the reader has undoubtedly asked why is learning a function possible at all? Certainly, for example, there are an uncountable number of different functions having values that agree with the four samples shown in Fig. 1.3. Why would a learning procedure happen to select the quadratic one shown in that figure? In order to make that selection we had at least to limit *a priori* the set of hypotheses to quadratic functions and then to insist that the one we chose passed through all four sample points. This kind of *a priori* information is called *bias*, and useful learning without bias is impossible.

We can gain more insight into the role of bias by considering the special case of learning a Boolean function of n dimensions. There are 2^n different Boolean inputs possible. Suppose we had no bias; that is $\mathcal{H}$ is the set of *all* 2^{2^n} Boolean functions, and we have no preference among those that fit

the samples in the training set. In this case, after being presented with one member of the training set and its value we can rule out precisely one-half of the members of $\mathcal{H}$—those Boolean functions that would misclassify this labeled sample. The remaining functions constitute what is called a "version space;" we'll explore that concept in more detail later. As we present more members of the training set, the graph of the number of hypotheses not yet ruled out as a function of the number of different patterns presented is as shown in Fig. 1.4. At any stage of the process, half of the remaining Boolean functions have value 1 and half have value 0 for *any* training pattern not yet seen. No generalization is possible in this case because the training patterns give no clue about the value of a pattern not yet seen. Only memorization is possible here, which is a trivial sort of learning.

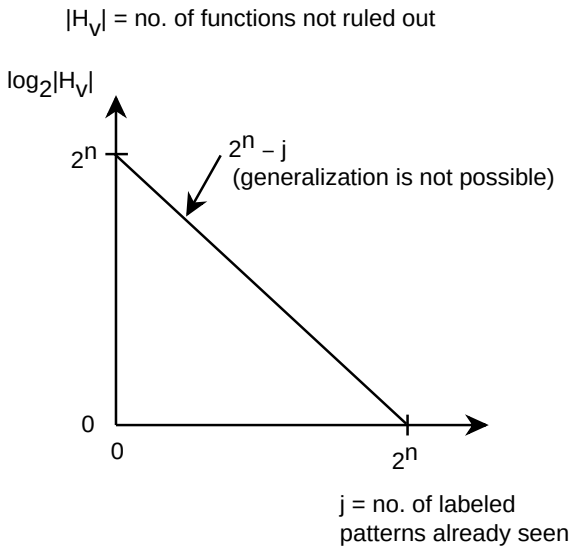


Figure 1.4: Hypotheses Remaining as a Function of Labeled Patterns Presented

But suppose we limited $\mathcal{H}$ to some subset, $\mathcal{H}_c$, of all Boolean functions. Depending on the subset and on the order of presentation of training patterns, a curve of hypotheses not yet ruled out might look something like the one shown in Fig. 1.5. In this case it is even possible that after seeing fewer than all 2^n labeled samples, there might be only one hypothesis that agrees with the training set. Certainly, even if there is more than one hypothesis

remaining, *most* of them may have the same value for *most* of the patterns not yet seen! The theory of *Probably Approximately Correct (PAC)* learning makes this intuitive idea precise. We'll examine that theory later.

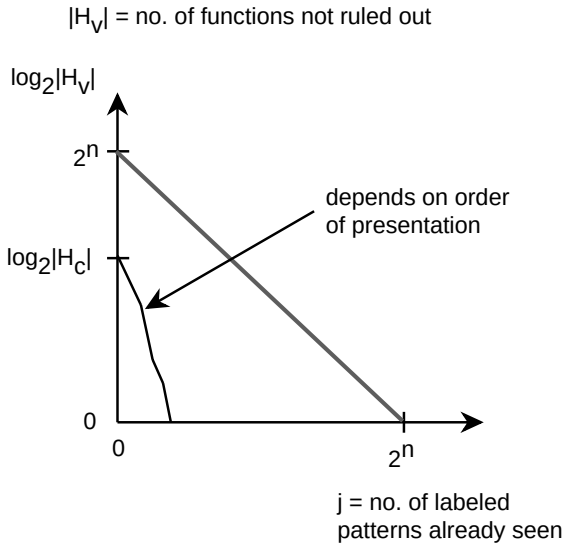


Figure 1.5: Hypotheses Remaining From a Restricted Subset

Let's look at a specific example of how bias aids learning. A Boolean function can be represented by a hypercube each of whose vertices represents a different input pattern. We show a 3-dimensional version in Fig. 1.6. There, we show a training set of six sample patterns and have marked those having a value of 1 by a small square and those having a value of 0 by a small circle. If the hypothesis set consists of just the *linearly separable* functions—those for which the positive and negative instances can be separated by a linear surface, then there is only one function remaining in this hypothsis set that is consistent with the training set. So, in this case, even though the training set does not contain all possible patterns, we can already pin down what the function must be—given the bias.

Machine learning researchers have identified two main varieties of bias, absolute and preference. In *absolute bias* (also called *restricted hypothesis-space bias*), one restricts $\mathcal{H}$ to a definite subset of functions. In our example of Fig. 1.6, the restriction was to linearly separable Boolean functions. In *preference bias*, one selects that hypothesis that is minimal according to

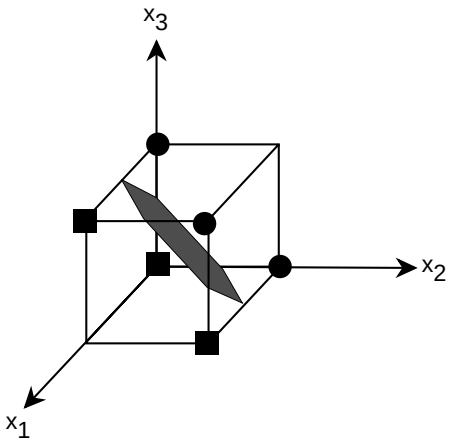


Figure 1.6: A Training Set That Completely Determines a Linearly Separable Function

some ordering scheme over all hypotheses. For example, if we had some way of measuring the *complexity* of a hypothesis, we might select the one that was simplest among those that performed satisfactorily on the training set. The principle of *Occam's razor*, used in science to prefer simple explanations to more complex ones, is a type of preference bias. (William of Occam, 1285-?1349, was an English philosopher who said: "*non sunt multiplicanda entia praeter necessitatem*," which means "entities should not be multiplied unnecessarily.")

1.4 Sample Applications

Our main emphasis in this book is on the concepts of machine learning—not on its applications. Nevertheless, if these concepts were irrelevant to real-world problems they would probably not be of much interest. As motivation, we give a short summary of some areas in which machine learning techniques have been successfully applied. [Langley, 1992] cites some of the following applications and others:

a. Rule discovery using a variant of ID3 for a printing industry problem [Evans & Fisher, 1992].

b. Electric power load forecasting using a k-nearest-neighbor rule system [Jabbour, K., *et al.*, 1987].

c. Automatic "help desk" assistant using a nearest-neighbor system [Acorn & Walden, 1992].

d. Planning and scheduling for a steel mill using ExpertEase, a marketed (ID3-like) system [Michie, 1992].

e. Classification of stars and galaxies [Fayyad, *et al.*, 1993].

Many application-oriented papers are presented at the annual conferences on Neural Information Processing Systems. Among these are papers on: speech recognition, dolphin echo recognition, image processing, bioengineering, diagnosis, commodity trading, face recognition, music composition, optical character recognition, and various control applications [Various Editors, 1989-1994].

As additional examples, [Hammerstrom, 1993] mentions:

a. Sharp's Japanese kanji character recognition system processes 200 characters per second with 99+% accuracy. It recognizes 3000+ characters.

b. NeuroForecasting Centre's (London Business School and University College London) trading strategy selection network earned an average annual profit of 18% against a conventional system's 12.3%.

c. Fujitsu's (plus a partner's) neural network for monitoring a continuous steel casting operation has been in successful operation since early 1990.

In summary, it is rather easy nowadays to find applications of machine learning techniques. This fact should come as no surprise inasmuch as many machine learning techniques can be viewed as extensions of well known statistical methods which have been successfully applied for many years.

1.5 Sources

Besides the rich literature in machine learning (a small part of which is referenced in the Bibliography), there are several textbooks that are worth mentioning [Hertz, Krogh, & Palmer, 1991, Weiss & Kulikowski, 1991, Natarjan, 1991, Fu, 1994, Langley, 1996]. [Shavlik & Dietterich, 1990,

Buchanan & Wilkins, 1993] are edited volumes containing some of the most important papers. A survey paper by [Dietterich, 1990] gives a good overview of many important topics. There are also well established conferences and publications where papers are given and appear including:

- The Annual Conferences on Advances in Neural Information Processing Systems
- The Annual Workshops on Computational Learning Theory
- The Annual International Workshops on Machine Learning
- The Annual International Conferences on Genetic Algorithms

  (The Proceedings of the above-listed four conferences are published by Morgan Kaufmann.)
- The journal *Machine Learning* (published by Kluwer Academic Publishers).

There is also much information, as well as programs and datasets, available over the Internet through the World Wide Web.

1.6 Bibliographical and Historical Remarks

To be added. Every chapter will contain a brief survey of the history of the material covered in that chapter.

Chapter 2

Boolean Functions

2.1 Representation

2.1.1 Boolean Algebra

Many important ideas about learning of functions are most easily presented using the special case of Boolean functions. There are several important subclasses of Boolean functions that are used as hypothesis classes for function learning. Therefore, we digress in this chapter to present a review of Boolean functions and their properties. (For a more thorough treatment see, for example, [Unger, 1989].)

A Boolean function, $f(x_1, x_2, \ldots, x_n)$ maps an n-tuple of (0,1) values to $\{0,1\}$. *Boolean algebra* is a convenient notation for representing Boolean functions. Boolean algebra uses the connectives $\cdot$, $+$, and $\bar{}$. For example, the *and* function of two variables is written $x_1 \cdot x_2$. By convention, the connective, "$\cdot$" is usually suppressed, and the *and* function is written x_1x_2. x_1x_2 has value 1 if and only if *both* x_1 and x_2 have value 1; if either x_1 or x_2 has value 0, x_1x_2 has value 0. The (inclusive) *or* function of two variables is written $x_1 + x_2$. $x_1 + x_2$ has value 1 if and only if either or both of x_1 or x_2 has value 1; if both x_1 and x_2 have value 0, $x_1 + x_2$ has value 0. The *complement* or *negation* of a variable, x, is written $\overline{x}$. $\overline{x}$ has value 1 if and only if x has value 0; if x has value 1, $\overline{x}$ has value 0.

These definitions are compactly given by the following rules for Boolean algebra:

$1 + 1 = 1$, $1 + 0 = 1$, $0 + 0 = 0$,

$1 \cdot 1 = 1$, $1 \cdot 0 = 0$, $0 \cdot 0 = 0$, and

$\overline{1} = 0$, $\overline{0} = 1$.

Sometimes the arguments and values of Boolean functions are expressed in terms of the constants T (*True*) and F (*False*) instead of 1 and 0, respectively.

The connectives $\cdot$ and $+$ are each commutative and associative. Thus, for example, $x_1(x_2x_3) = (x_1x_2)x_3$, and both can be written simply as $x_1x_2x_3$. Similarly for $+$.

A Boolean formula consisting of a single variable, such as x_1 is called an *atom*. One consisting of either a single variable or its complement, such as $\overline{x_1}$, is called a *literal*.

The operators $\cdot$ and $+$ do not commute between themselves. Instead, we have DeMorgan's laws (which can be verified by using the above definitions):

$\overline{x_1x_2} = \overline{x_1} + \overline{x_2}$, and
$\overline{x_1 + x_2} = \overline{x_1}\ \overline{x_2}$.

2.1.2 Diagrammatic Representations

We saw in the last chapter that a Boolean function could be represented by labeling the vertices of a cube. For a function of n variables, we would need an n-dimensional *hypercube*. In Fig. 2.1 we show some 2- and 3-dimensional examples. Vertices having value 1 are labeled with a small square, and vertices having value 0 are labeled with a small circle.

Using the hypercube representations, it is easy to see how many Boolean functions of n dimensions there are. A 3-dimensional cube has $2^3 = 8$ vertices, and each may be labeled in two different ways; thus there are $2^{(2^3)} = 256$ different Boolean functions of 3 variables. In general, there are 2^{2^n} Boolean functions of n variables.

We will be using 2- and 3-dimensional cubes later to provide some intuition about the properties of certain Boolean functions. Of course, we cannot visualize hypercubes (for $n > 3$), and there are many surprising properties of higher dimensional spaces, so we must be careful in using intuitions gained in low dimensions. One diagrammatic technique for dimensions slightly higher than 3 is the *Karnaugh map*. A Karnaugh map is an array of values of a Boolean function in which the horizontal rows are indexed by the values of some of the variables and the vertical columns are indexed by the rest. The rows and columns are arranged in such a way that entries that are adjacent in the map correspond to vertices that are adjacent in the hypercube representation. We show an example of the 4-dimensional even parity function in Fig. 2.2. (An *even parity function* is

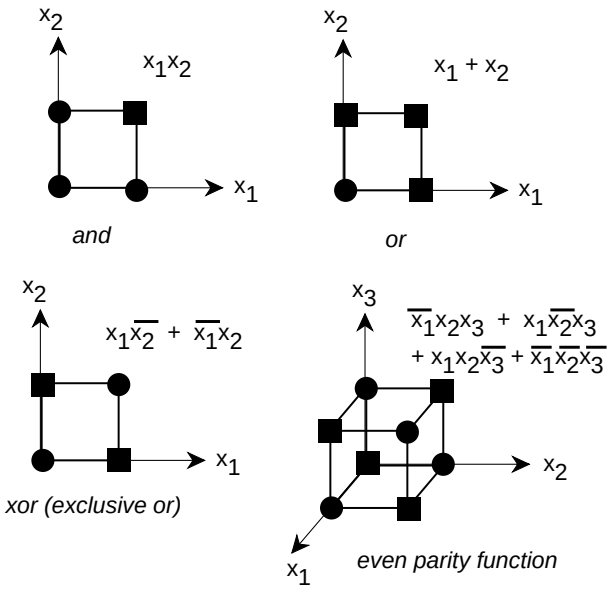


Figure 2.1: Representing Boolean Functions on Cubes

a Boolean function that has value 1 if there are an even number of its arguments that have value 1; otherwise it has value 0.) Note that all adjacent cells in the table correspond to inputs differing in only one component.

Also describe *general logic diagrams*, [Wnek, *et al.*, 1990].

2.2 Classes of Boolean Functions

2.2.1 Terms and Clauses

To use absolute bias in machine learning, we limit the class of hypotheses. In learning Boolean functions, we frequently use some of the common subclasses of those functions. Therefore, it will be important to know about these subclasses.

One basic subclass is called *terms*. A term is any function written in the form $l_1 l_2 \cdots l_k$, where the l_i are literals. Such a form is called a *conjunction* of literals. Some example terms are $x_1 x_7$ and $x_1 x_2 \overline{x_4}$. The *size* of a term is the number of literals it contains. The examples are of sizes 2 and 3, respectively. (Strictly speaking, the *class* of conjunctions of literals

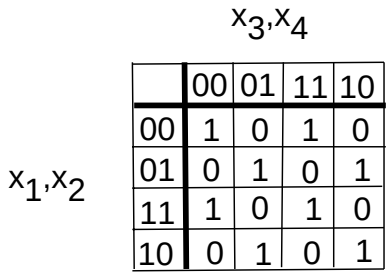


Figure 2.2: A Karnaugh Map

is called the *monomials*, and a conjunction of literals itself is called a *term*. This distinction is a fine one which we elect to blur here.)

It is easy to show that there are exactly 3^n possible terms of n variables. The number of terms of size k or less is bounded from above by $\sum_{i=0}^{k} C(2n, i) = O(n^k)$, where $C(i, j) = \frac{i!}{(i-j)!j!}$ is the binomial coefficient.

Probably I'll put in a simple term-learning algorithm here—so we can get started on learning! Also for DNF functions and decision lists—as they are defined in the next few pages.

A *clause* is any function written in the form $l_1 + l_2 + \cdots + l_k$, where the l_i are literals. Such a form is called a *disjunction* of literals. Some example clauses are $x_3 + x_5 + x_6$ and $x_1 + \overline{x_4}$. The *size* of a clause is the number of literals it contains. There are 3^n possible clauses and fewer than $\sum_{i=0}^{k} C(2n, i)$ clauses of size k or less. If f is a term, then (by De Morgan's laws) $\overline{f}$ is a clause, and vice versa. Thus, terms and clauses are duals of each other.

In psychological experiments, conjunctions of literals seem easier for humans to learn than disjunctions of literals.

2.2.2 DNF Functions

A Boolean function is said to be in *disjunctive normal form (DNF)* if it can be written as a *disjunction* of terms. Some examples in DNF are: $f = x_1x_2 + x_2x_3x_4$ and $f = x_1\overline{x_3} + \overline{x_2}\ \overline{x_3} + x_1x_2\overline{x_3}$. A DNF expression is called a k-term DNF expression if it is a disjunction of k terms; it is in the class k-DNF if the size of its largest term is k. The examples above are 2-term and 3-term expressions, respectively. Both expressions are in the class 3-DNF.

Each term in a DNF expression for a function is called an *implicant* because it "implies" the function (if the term has value 1, so does the

function). In general, a term, t, is an implicant of a function, f, if f has value 1 whenever t does. A term, t, is a *prime implicant* of f if the term, t', formed by taking any literal out of an implicant t is no longer an implicant of f. (The implicant cannot be "divided" by any term and remain an implicant.)

Thus, both $x_2\overline{x_3}$ and $\overline{x_1}\ \overline{x_3}$ are prime implicants of $f = x_2\overline{x_3} + \overline{x_1}\ \overline{x_3} + x_2x_1\overline{x_3}$, but $x_2x_1\overline{x_3}$ is not.

The relationship between implicants and prime implicants can be geometrically illustrated using the cube representation for Boolean functions. Consider, for example, the function $f = x_2\overline{x_3} + \overline{x_1}\ \overline{x_3} + x_2x_1\overline{x_3}$. We illustrate it in Fig. 2.3. Note that each of the three planes in the figure "cuts off" a group of vertices having value 1, but none cuts off any vertices having value 0. These planes are pictorial devices used to isolate certain lower dimensional *subfaces* of the cube. Two of them isolate one-dimensional *edges*, and the third isolates a zero-dimensional *vertex*. Each group of vertices on a subface corresponds to one of the implicants of the function, f, and thus each implicant corresponds to a subface of some dimension. A k-dimensional subface corresponds to an $(n-k)$-size implicant term. The function is written as the disjunction of the implicants—corresponding to the union of all the vertices cut off by all of the planes. Geometrically, an implicant is prime if and only if its corresponding subface is the largest dimensional subface that includes all of its vertices and no other vertices having value 0. Note that the term $x_2x_1\overline{x_3}$ is not a prime implicant of f. (In this case, we don't even have to include this term in the function because the vertex cut off by the plane corresponding to $x_2x_1\overline{x_3}$ is already cut off by the plane corresponding to $x_2\overline{x_3}$.) The other two implicants are prime because their corresponding subfaces cannot be expanded without including vertices having value 0.

Note that all Boolean functions can be represented in DNF—trivially by disjunctions of terms of size n where each term corresponds to one of the vertices whose value is 1. Whereas there are 2^{2^n} functions of n dimensions in DNF (since any Boolean function can be written in DNF), there are just $2^{O(n^k)}$ functions in k-DNF.

All Boolean functions can also be represented in DNF in which each term is a prime implicant, but that representation is not unique, as shown in Fig. 2.4.

If we can express a function in DNF form, we can use the *consensus* method to find an expression for the function in which each term is a prime implicant. The consensus method relies on two results:

We may replace this section with one describing the Quine-McCluskey method instead.

- Consensus:

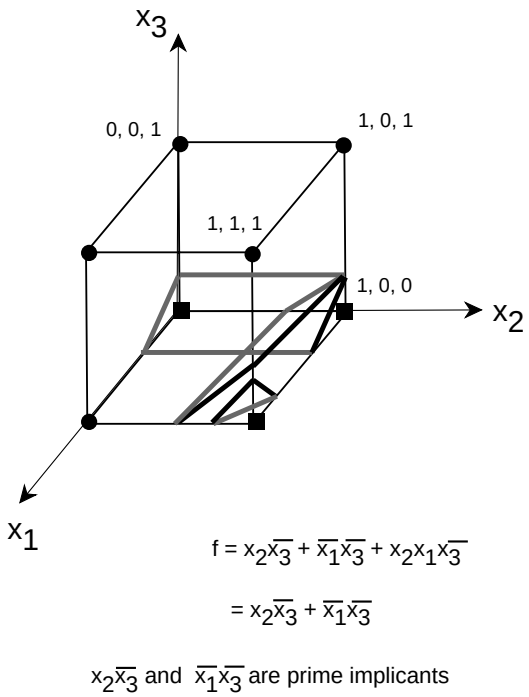


Figure 2.3: A Function and its Implicants

$$x_i \cdot f_1 + \overline{x_i} \cdot f_2 = x_i \cdot f_1 + \overline{x_i} \cdot f_2 + f_1 \cdot f_2$$

where f_1 and f_2 are terms such that no literal appearing in f_1 appears complemented in f_2. $f_1 \cdot f_2$ is called the *consensus* of $x_i \cdot f_1$ and $\overline{x_i} \cdot f_2$. Readers familiar with the *resolution* rule of inference will note that consensus is the dual of resolution.

Examples: x_1 is the consensus of $x_1 x_2$ and $x_1 \overline{x_2}$. The terms $\overline{x_1} x_2$ and $x_1 \overline{x_2}$ have no consensus since each term has more than one literal appearing complemented in the other.

- Subsumption:

$$x_i \cdot f_1 + f_1 = f_1$$

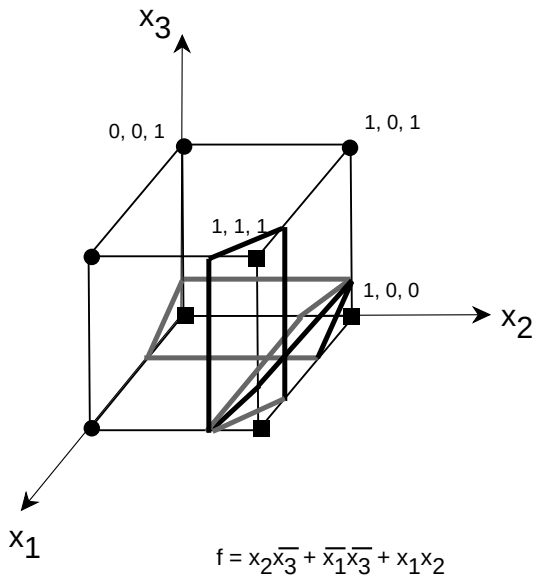


Figure 2.4: Non-Uniqueness of Representation by Prime Implicants

where f_1 is a term. We say that f_1 *subsumes* $x_i \cdot f_1$.

Example: $\overline{x_1}\ \overline{x_4}x_5$ subsumes $\overline{x_1}\ \overline{x_4}\ \overline{x_2}x_5$

The consensus method for finding a set of prime implicants for a function, f, iterates the following operations on the terms of a DNF expression for f until no more such operations can be applied:

a. initialize the process with the set, $\mathcal{T}$, of terms in the DNF expression of f,

b. compute the consensus of a pair of terms in $\mathcal{T}$ and add the result to $\mathcal{T}$,

c. eliminate any terms in $\mathcal{T}$ that are subsumed by other terms in $\mathcal{T}$.

When this process halts, the terms remaining in $\mathcal{T}$ are all prime implicants of f.

Example: Let $f = \overline{x_1}x_2 + \overline{x_1}\,\overline{x_2}x_3 + \overline{x_1}\,\overline{x_2}\,\overline{x_3}\,\overline{x_4}x_5$. We show a derivation of a set of prime implicants in the *consensus tree* of Fig. 2.5. The circled numbers adjoining the terms indicate the order in which the consensus and subsumption operations were performed. Shaded boxes surrounding a term indicate that it was subsumed. The final form of the function in which all terms are prime implicants is: $f = \overline{x_1}x_2 + \overline{x_1}x_3 + \overline{x_1}\,\overline{x_4}x_5$. Its terms are all of the non-subsumed terms in the consensus tree.

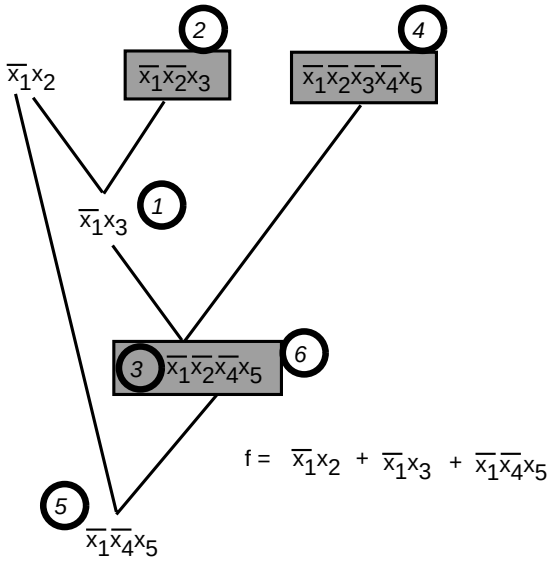


Figure 2.5: A Consensus Tree

2.2.3 CNF Functions

Disjunctive normal form has a dual: *conjunctive normal form (CNF)*. A Boolean function is said to be in CNF if it can be written as a *conjunction* of clauses. An example in CNF is: $f = (x_1 + x_2)(x_2 + x_3 + x_4)$. A CNF expression is called a k-clause CNF expression if it is a conjunction of k clauses; it is in the class k-CNF if the size of its largest clause is k. The example is a 2-clause expression in 3-CNF. If f is written in DNF, an

application of De Morgan's law renders $\overline{f}$ in CNF, and vice versa. Because CNF and DNF are duals, there are also $2^{O(n^k)}$ functions in k-CNF.

2.2.4 Decision Lists

Rivest has proposed a class of Boolean functions called *decision lists* [Rivest, 1987]. A decision list is written as an ordered list of pairs:

(t_q, v_q)

(t_{q-1}, v_{q-1})

$\ldots$

(t_i, v_i)

$\ldots$

(t_2, v_2)

(T, v_1)

where the v_i are either 0 or 1, the t_i are terms in $(x_1, \ldots, x_n)$, and T is a term whose value is 1 (regardless of the values of the x_i). The value of a decision list is the value of v_i for the first t_i in the list that has value 1. (At least one t_i will have value 1, because the last one does; v_1 can be regarded as a *default* value of the decision list.) The decision list is of *size* k, if the size of the largest term in it is k. The class of decision lists of size k or less is called k-DL.

An example decision list is:

$f =$

$(\overline{x_1}x_2, 1)$

$(\overline{x_1}\ \overline{x_2}x_3, 0)$

$\overline{x_2}x_3, 1)$

$(1, 0)$

f has value 0 for $x_1 = 0$, $x_2 = 0$, and $x_3 = 1$. It has value 1 for $x_1 = 1$, $x_2 = 0$, and $x_3 = 1$. This function is in 3-DL.

It has been shown that the class k-DL is a strict superset of the union of k-DNF and k-CNF. There are $2^{O[n^k k \log(n)]}$ functions in k-DL [Rivest, 1987].

Interesting generalizations of decision lists use other Boolean functions in place of the terms, t_i. For example we might use linearly separable functions in place of the t_i (see below and [Marchand & Golea, 1993]).

2.2.5 Symmetric and Voting Functions

A Boolean function is called *symmetric* if it is invariant under permutations of the input variables. For example, any function that is dependent only on the number of input variables whose values are 1 is a symmetric function. The *parity* functions, which have value 1 depending on whether or not the number of input variables with value 1 is even or odd is a symmetric function. (The *exclusive or* function, illustrated in Fig. 2.1, is an odd-parity function of two dimensions. The *or* and *and* functions of two dimensions are also symmetric.)

An important subclass of the symmetric functions is the class of *voting functions* (also called m-of-n functions). A k-voting function has value 1 if and only if k or more of its n inputs has value 1. If $k = 1$, a voting function is the same as an n-sized clause; if $k = n$, a voting function is the same as an n-sized term; if $k = (n + 1)/2$ for n odd or $k = 1 + n/2$ for n even, we have the *majority* function.

2.2.6 Linearly Separable Functions

The linearly separable functions are those that can be expressed as follows:

$$f = \text{thresh}(\sum_{i=1}^{n} w_i x_i, \theta)$$

where w_i, $i = 1, \ldots, n$, are real-valued numbers called *weights*, θ is a real-valued number called the *threshold*, and $\text{thresh}(\sigma, \theta)$ is 1 if $\sigma \geq \theta$ and 0 otherwise. (Note that the concept of linearly separable functions can be extended to non-Boolean inputs.) The k-voting functions are all members of the class of linearly separable functions in which the weights all have unit value and the threshold depends on k. Thus, terms and clauses are special cases of linearly separable functions.

A convenient way to write linearly separable functions uses vector notation:

$$f = \text{thresh}(\mathbf{X} \cdot \mathbf{W}, \theta)$$

where $\mathbf{X} = (x_1, \ldots, x_n)$ is an n-dimensional vector of input variables, $\mathbf{W} = (w_1, \ldots, w_n)$ is an n-dimensional vector of weight values, and $\mathbf{X} \cdot \mathbf{W}$ is the *dot* (or *inner*) product of the two vectors. Input vectors for which f has value 1 lie in a half-space on one side of (and on) a hyperplane whose orientation is normal to $\mathbf{W}$ and whose position (with respect to the origin)

is determined by θ. We saw an example of such a separating plane in Fig. 1.6. With this idea in mind, it is easy to see that two of the functions in Fig. 2.1 are linearly separable, while two are not. Also note that the terms in Figs. 2.3 and 2.4 are linearly separable functions as evidenced by the separating planes shown.

There is no closed-form expression for the number of linearly separable functions of n dimensions, but the following table gives the numbers for n up to 6.

n	Boolean Functions	Linearly Separable Functions
1	4	4
2	16	14
3	256	104
4	65,536	1,882
5	$\approx 4.3 \times 10^9$	94,572
6	$\approx 1.8 \times 10^{19}$	15,028,134

[Muroga, 1971] has shown that (for $n > 1$) there are no more than 2^{n^2} linearly separable functions of n dimensions. (See also [Winder, 1961, Winder, 1962].)

2.3 Summary

The diagram in Fig. 2.6 shows some of the set inclusions of the classes of Boolean functions that we have considered. We will be confronting these classes again in later chapters.

The sizes of the various classes are given in the following table (adapted from [Dietterich, 1990, page 262]):

Class	Size of Class
terms	3^n
clauses	3^n
k-term DNF	$2^{O(kn)}$
k-clause CNF	$2^{O(kn)}$
k-DNF	$2^{O(n^k)}$
k-CNF	$2^{O(n^k)}$
k-DL	$2^{O[n^k k \log(n)]}$
lin sep	$2^{O(n^2)}$
DNF	2^{2^n}

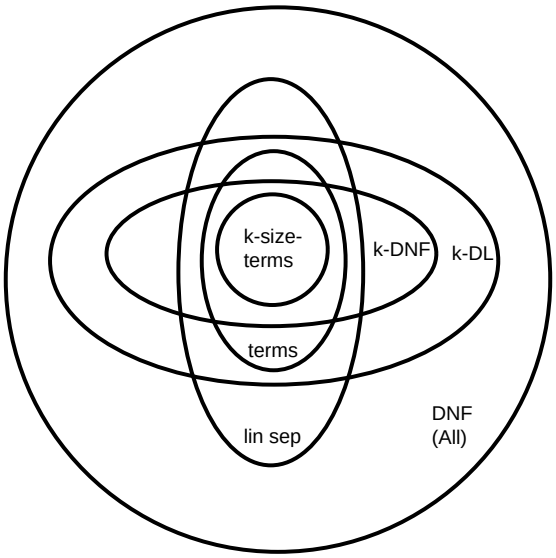


Figure 2.6: Classes of Boolean Functions

2.4 Bibliographical and Historical Remarks

To be added.

Chapter 3

Using Version Spaces for Learning

3.1 Version Spaces and Mistake Bounds

The first learning methods we present are based on the concepts of *version spaces* and *version graphs*. These ideas are most clearly explained for the case of Boolean function learning. Given an initial hypothesis set $\mathcal{H}$ (a subset of all Boolean functions) and the values of $f(\mathbf{X})$ for each $\mathbf{X}$ in a training set, Ξ, the version space is that subset of hypotheses, $\mathcal{H}_v$, that is consistent with these values. A hypothesis, h, is *consistent* with the values of $\mathbf{X}$ in Ξ if and only if $h(\mathbf{X}) = f(\mathbf{X})$ for all $\mathbf{X}$ in Ξ. We say that the hypotheses in $\mathcal{H}$ that are not consistent with the values in the training set are *ruled out* by the training set.

We could imagine (conceptually only!) that we have devices for implementing every function in $\mathcal{H}$. An incremental training procedure could then be defined which presented each pattern in Ξ to each of these functions and then eliminated those functions whose values for that pattern did not agree with its given value. At any stage of the process we would then have left some subset of functions that are consistent with the patterns presented so far; this subset is the version space for the patterns already presented. This idea is illustrated in Fig. 3.1.

Consider the following procedure for classifying an arbitrary input pattern, $\mathbf{X}$: the pattern is put in the same class (0 or 1) as are the majority of the outputs of the functions in the version space. During the learning procedure, if this majority is not equal to the value of the pattern presented,

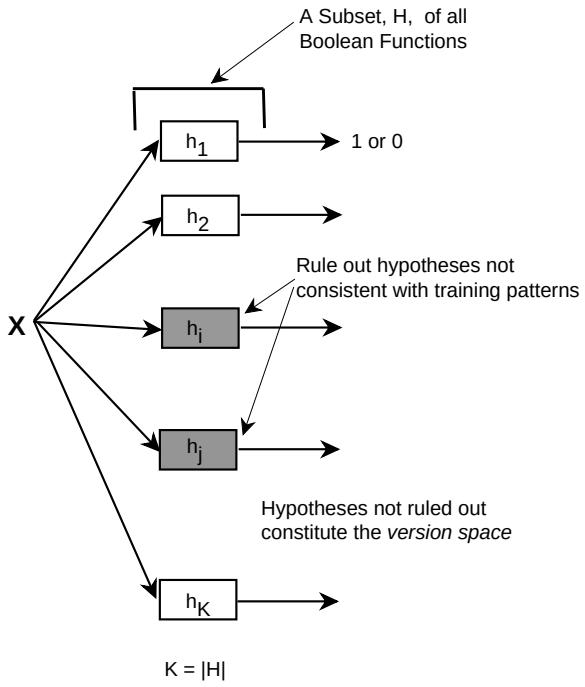


Figure 3.1: Implementing the Version Space

we say a *mistake* is made, and we revise the version space accordingly—eliminating all those (majority of the) functions voting incorrectly. Thus, whenever a mistake is made, we rule out at least half of the functions remaining in the version space.

How many mistakes can such a procedure make? Obviously, we can make no more than $\log_2(|\mathcal{H}|)$ mistakes, where $|\mathcal{H}|$ is the number of hypotheses in the original hypothesis set, $\mathcal{H}$. (Note, though, that the number of training patterns seen before this maximum number of mistakes is made might be much greater.) This theoretical (and very impractical!) result (due to [Littlestone, 1988]) is an example of a *mistake bound*—an important concept in machine learning theory. It shows that there must exist a learning procedure that makes no more mistakes than this upper bound. Later, we'll derive other mistake bounds.

As a special case, if our bias was to limit $\mathcal{H}$ to terms, we would make no more than $\log_2(3^n) = n\log_2(3) = 1.585n$ mistakes before *exhausting* the

version space. This result means that if f were a term, we would make no more than $1.585n$ mistakes before learning f, and otherwise we would make no more than that number of mistakes before being able to decide that f is not a term.

Even if we do not have sufficient training patterns to reduce the version space to a single function, it may be that there are enough training patterns to reduce the version space to a set of functions such that most of them assign the same values to most of the patterns we will see henceforth. We could select one of the remaining functions at random and be reasonably assured that it will generalize satisfactorily. We next discuss a computationally more feasible method for representing the version space.

3.2 Version Graphs

Boolean functions can be ordered by *generality*. A Boolean function, f_1, is *more general* than a function, f_2, (and f_2 is *more specific* than f_1), if f_1 has value 1 for all of the arguments for which f_2 has value 1, and $f_1 \neq f_2$. For example, x_3 is more general than x_2x_3 but is not more general than $x_3 + x_2$.

We can form a graph with the hypotheses, $\{h_i\}$, in the version space as nodes. A node in the graph, h_i, has an arc directed to node, h_j, if and only if h_j is more general than h_i. We call such a graph a *version graph*. In Fig. 3.2, we show an example of a version graph over a 3-dimensional input space for hypotheses restricted to terms (with none of them yet ruled out).

That function, denoted here by "1," which has value 1 for all inputs, corresponds to the node at the top of the graph. (It is more general than any other term.) Similarly, the function "0" is at the bottom of the graph. Just below "1" is a row of nodes corresponding to all terms having just one literal, and just below them is a row of nodes corresponding to terms having two literals, and so on. There are $3^3 = 27$ functions altogether (the function "0," included in the graph, is technically not a term). To make our portrayal of the graph less cluttered only some of the arcs are shown; each node in the actual graph has an arc directed to all of the nodes above it that are more general.

We use this same example to show how the version graph changes as we consider a set of labeled samples in a training set, Ξ. Suppose we first consider the training pattern (1, 0, 1) with value 0. Some of the functions in the version graph of Fig. 3.2 are inconsistent with this training pattern. These ruled out nodes are no longer in the version graph and are

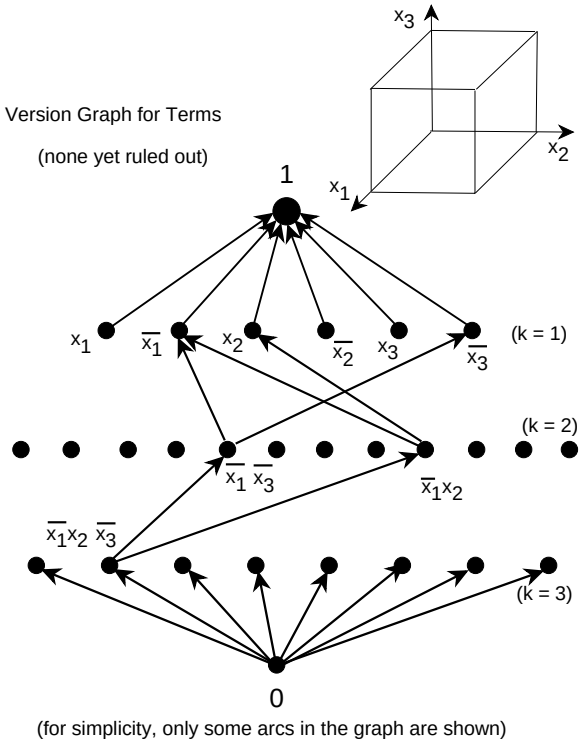


Figure 3.2: A Version Graph for Terms

shown shaded in Fig. 3.3. We also show there the three-dimensional cube representation in which the vertex (1, 0, 1) has value 0.

In a version graph, there are always a set of hypotheses that are maximally general and a set of hypotheses that are maximally specific. These are called the *general boundary set (gbs)* and the *specific boundary set (sbs)*, respectively. In Fig. 3.4, we have the version graph as it exists after learning that (1,0,1) has value 0 and (1, 0, 0) has value 1. The gbs and sbs are shown.

Boundary sets are important because they provide an alternative to representing the entire version space explicitly, which would be impractical. Given only the boundary sets, it is possible to determine whether or not any hypothesis (in the prescribed class of Boolean functions we are using)

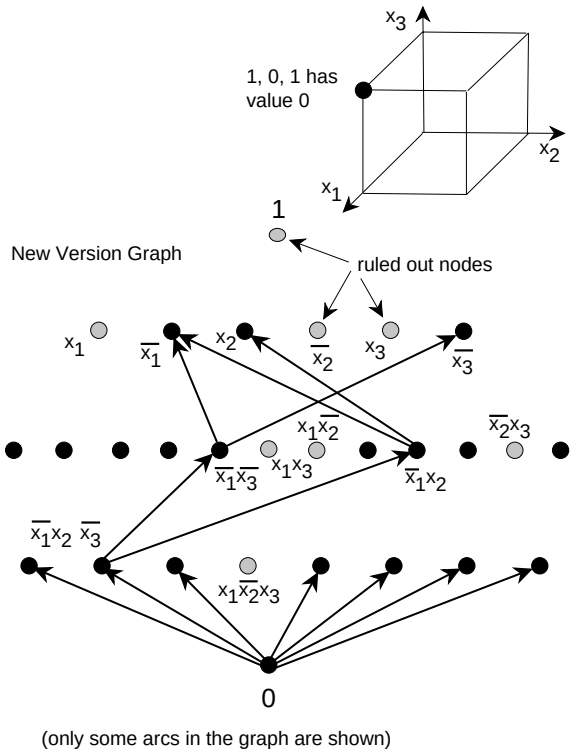


Figure 3.3: The Version Graph Upon Seeing (1, 0, 1)

is a member or not of the version space. This determination is possible because of the fact that any member of the version space (that is not a member of one of the boundary sets) is more specific than some member of the general boundary set and is more general than some member of the specific boundary set.

If we limit our Boolean functions that can be in the version space to terms, it is a simple matter to determine maximally general and maximally specific functions (assuming that there is some term that is in the version space). A maximally specific one corresponds to a subface of *minimal dimension* that contains all the members of the training set labelled by a 1 and no members labelled by a 0. A maximally general one corresponds to a subface of *maximal dimension* that contains all the members of the training

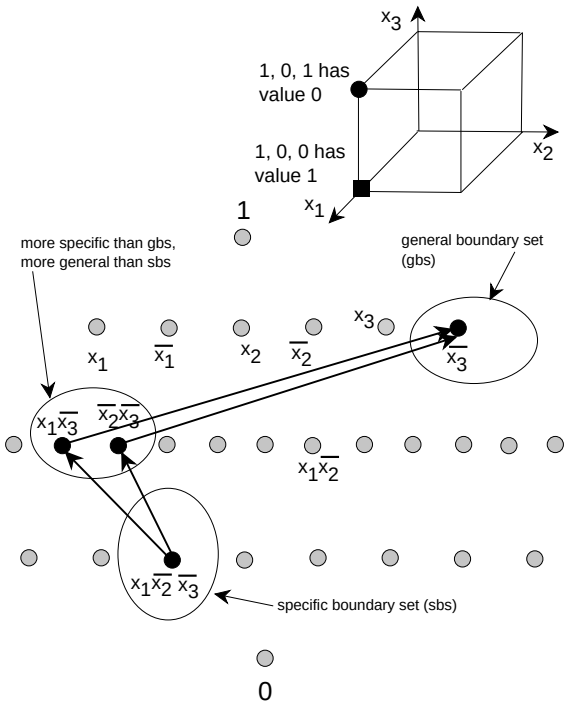


Figure 3.4: The Version Graph Upon Seeing (1, 0, 1) and (1, 0, 0)

set labelled by a 1 and no members labelled by a 0. Looking at Fig. 3.4, we see that the subface of minimal dimension that contains (1, 0, 0) but does not contain (1, 0, 1) is just the vertex (1, 0, 0) itself—corresponding to the function $x_1\overline{x_2}\ \overline{x_3}$. The subface of maximal dimension that contains (1, 0, 0) but does not contain (1, 0, 1) is the bottom face of the cube—corresponding to the function $\overline{x_3}$. In Figs. 3.2 through 3.4 the sbs is always singular. Version spaces for terms always have singular specific boundary sets. As seen in Fig. 3.3, however, the gbs of a version space for terms need not be singular.

3.3 Learning as Search of a Version Space

[To be written. Relate to term learning algorithm presented in Chapter Two. Also discuss best-first search methods. See Pat Langley's example

using "pseudo-cells" of how to generate and eliminate hypotheses.]

Selecting a hypothesis from the version space can be thought of as a search problem. One can start with a very general function and specialize it through various specialization operators until one finds a function that is consistent (or adequately so) with a set of training patterns. Such procedures are usually called *top-down* methods. Or, one can start with a very special function and generalize it—resulting in *bottom-up* methods. We shall see instances of both styles of learning in this book.

Compare this view of top-down versus bottom-up with the *divide-and-conquer* and the *covering* (or AQ) methods of decision-tree induction.

3.4 The Candidate Elimination Method

The *candidate elimination method*, is an incremental method for computing the boundary sets. Quoting from [Hirsh, 1994, page 6]:

> "The *candidate-elimination algorithm* manipulates the boundary-set representation of a version space to create boundary sets that represent a new version space consistent with all the previous instances plus the new one. For a positive exmple the algorithm generalizes the elements of the [sbs] as little as possible so that they cover the new instance yet remain consistent with past data, and removes those elements of the [gbs] that do not cover the new instance. For a negative instance the algorithm specializes elements of the [gbs] so that they no longer cover the new instance yet remain consistent with past data, and removes from the [sbs] those elements that mistakenly cover the new, negative instance."

The method uses the following definitions (adapted from [Genesereth & Nilsson, 1987]):

- a hypothesis is called *sufficient* if and only if it has value 1 for all training samples labeled by a 1,
- a hypothesis is called *necessary* if and only if it has value 0 for all training samples labeled by a 0.

Here is how to think about these definitions: A hypothesis implements a *sufficient* condition that a training sample has value 1 if the hypothesis has value 1 for all of the positive instances; a hypothesis implements a *necessary* condition that a training sample has value 1 if the hypothesis has value 0 for all of the negative instances. A hypothesis is consistent with the training

set (and thus is in the version space) if and only if it is both sufficient and necessary.

We start (before receiving any members of the training set) with the function "0" as the singleton element of the specific boundary set and with the function "1" as the singleton element of the general boundary set. Upon receiving a new labeled input vector, the boundary sets are changed as follows:

a. If the new vector is labelled with a 1:

   The new general boundary set is obtained from the previous one by excluding any elements in it that are not sufficient. (That is, we exclude any elements that have value 0 for the new vector.)

   The new specific boundary set is obtained from the previous one by replacing each element, h_i, in it by all of its *least generalizations*.

   The hypothesis h_g is a *least generalization* of h if and only if: a) h is more specific than h_g, b) h_g is sufficient, c) no function (including h) that is more specific than h_g is sufficient, and d) h_g is more specific than some member of the new general boundary set. It might be that $h_g = h$. Also, least generalizations of two different functions in the specific boundary set may be identical.

b. If the new vector is labelled with a 0:

   The new specific boundary set is obtained from the previous one by excluding any elements in it that are not necessary. (That is, we exclude any elements that have value 1 for the new vector.)

   The new general boundary set is obtained from the previous one by replacing each element, h_i, in it by all of its *least specializations*.

   The hypothesis h_s is a *least specialization* of h if and only if: a) h is more general than h_s, b) h_s is necessary, c) no function (including h) that is more general than h_s is necessary, and d) h_s is more general than some member of the new specific boundary set. Again, it might be that $h_s = h$, and least specializations of two different functions in the general boundary set may be identical.

As an example, suppose we present the vectors in the following order:

vector	label
(1, 0, 1)	0
(1, 0, 0)	1
(1, 1, 1)	0
(0, 0, 1)	0

We start with general boundary set, "1", and specific boundary set, "0." After seeing the first sample, (1, 0, 1), labeled with a 0, the specific boundary set stays at "0" (it is necessary), and we change the general boundary set to $\{\overline{x_1}, x_2, \overline{x_3}\}$. Each of the functions, $\overline{x_1}$, x_2, and $\overline{x_3}$, are least specializations of "1" (they are necessary, "1" is not, they are more general than "0", and there are no functions that are more general than they and also necessary).

Then, after seeing (1, 0, 0), labeled with a 1, the general boundary set changes to $\{\overline{x_3}\}$ (because $\overline{x_1}$ and x_2 are not sufficient), and the specific boundary set is changed to $\{x_1\overline{x_2}\,\overline{x_3}\}$. This single function is a least generalization of "0" (it is sufficient, "0" is more specific than it, no function (including "0") that is more specific than it is sufficient, and it is more specific than some member of the general boundary set.

When we see (1, 1, 1), labeled with a 0, we do not change the specific boundary set because its function is still necessary. We do not change the general boundary set either because $\overline{x_3}$ is still necessary.

Finally, when we see (0, 0, 1), labeled with a 0, we do not change the specific boundary set because its function is still necessary. We do not change the general boundary set either because $\overline{x_3}$ is still necessary.

Maybe I'll put in an example of a version graph for non-Boolean functions.

3.5 Bibliographical and Historical Remarks

The concept of version spaces and their role in learning was first investigated by Tom Mitchell [Mitchell, 1982]. Although these ideas are not used in practical machine learning procedures, they do provide insight into the nature of hypothesis selection. In order to accomodate noisy data, version spaces have been generalized by [Hirsh, 1994] to allow hypotheses that are not necessarily consistent with the training set.

More to be added.

Chapter 4

Neural Networks

In chapter two we defined several important subsets of Boolean functions. Suppose we decide to use one of these subsets as a hypothesis set for supervised function learning. We next have the question of how best to implement the function as a device that gives the outputs prescribed by the function for arbitrary inputs. In this chapter we describe how networks of non-linear elements can be used to implement various input-output functions and how they can be trained using supervised learning methods.

Networks of non-linear elements, interconnected through adjustable weights, play a prominent role in machine learning. They are called *neural networks* because the non-linear elements have as their inputs a weighted sum of the outputs of other elements—much like networks of biological neurons do. These networks commonly use the threshold element which we encountered in chapter two in our study of linearly separable Boolean functions. We begin our treatment of neural nets by studying this threshold element and how it can be used in the simplest of all networks, namely ones composed of a single threshold element.

4.1 Threshold Logic Units

4.1.1 Definitions and Geometry

Linearly separable (threshold) functions are implemented in a straightforward way by summing the weighted inputs and comparing this sum to a threshold value as shown in Fig. 4.1. This structure we call a *threshold logic unit (TLU)*. Its output is 1 or 0 depending on whether or not

the weighted sum of its inputs is greater than or equal to a threshold value, θ. It has also been called an *Adaline* (for adaptive linear element) [Widrow, 1962, Widrow & Lehr, 1990], an LTU (linear threshold unit), a *perceptron*, and a *neuron*. (Although the word "perceptron" is often used nowadays to refer to a single TLU, Rosenblatt originally defined it as a class of *networks* of threshold elements [Rosenblatt, 1958].)

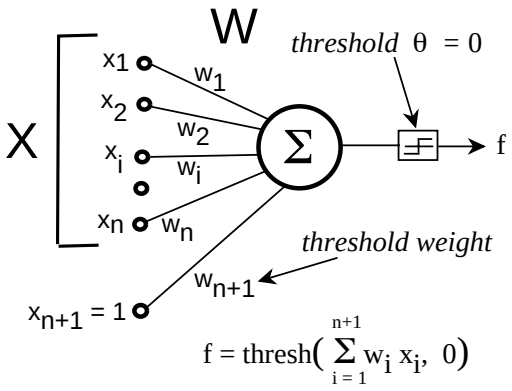


Figure 4.1: A Threshold Logic Unit (TLU)

The n-dimensional feature or input vector is denoted by $\mathbf{X} = (x_1, \ldots, x_n)$. When we want to distinguish among different feature vectors, we will attach subscripts, such as $\mathbf{X}_i$. The components of $\mathbf{X}$ can be any real-valued numbers, but we often specialize to the binary numbers 0 and 1. The weights of a TLU are represented by an n-dimensional *weight vector*, $\mathbf{W} = (w_1, \ldots, w_n)$. Its components are real-valued numbers (but we sometimes specialize to integers). The TLU has output 1 if $\sum_{i=1}^{n} x_i w_i \geq \theta$; otherwise it has output 0. The weighted sum that is calculated by the TLU can be simply represented as a vector dot product, $\mathbf{X} \bullet \mathbf{W}$. (If the pattern and weight vectors are thought of as "column" vectors, this dot product is then sometimes written as $\mathbf{X}^t\mathbf{W}$, where the "row" vector $\mathbf{X}^t$ is the transpose of $\mathbf{X}$.) Often, the threshold, θ, of the TLU is fixed at 0; in that case, arbitrary thresholds are achieved by using $(n+1)$-dimensional "augmented" vectors, $\mathbf{Y}$, and $\mathbf{V}$, whose first n components are the same as those of $\mathbf{X}$ and $\mathbf{W}$, respectively. The $(n+1)$-st component, x_{n+1}, of the augmented feature vector, $\mathbf{Y}$, always has value 1; the $(n+1)$-st component, w_{n+1}, of the augmented weight vector, $\mathbf{V}$, is set equal to the negative of the desired threshold value. (When we want to emphasize the use of

augmented vectors, we'll use the $\mathbf{Y}$,$\mathbf{V}$ notation; however when the context of the discussion makes it clear about what sort of vectors we are talking about, we'll lapse back into the more familiar $\mathbf{X}$,$\mathbf{W}$ notation.) In the $\mathbf{Y}$,$\mathbf{V}$ notation, the TLU has an output of 1 if $\mathbf{Y}\bullet\mathbf{V} \geq 0$. Otherwise, the output is 0.

We can give an intuitively useful geometric description of a TLU. A TLU divides the input space by a hyperplane as sketched in Fig. 4.2. The hyperplane is the boundary between patterns for which $\mathbf{X}\bullet\mathbf{W} + w_{n+1} > 0$ and patterns for which $\mathbf{X}\bullet\mathbf{W} + w_{n+1} < 0$. Thus, the equation of the hyperplane itself is $\mathbf{X}\bullet\mathbf{W} + w_{n+1} = 0$. The unit vector that is normal to the hyperplane is $\mathbf{n} = \frac{\mathbf{W}}{|\mathbf{W}|}$, where $|\mathbf{W}| = \sqrt{(w_1^2 + \ldots + w_n^2)}$ is the length of the vector $\mathbf{W}$. (The *normal form* of the hyperplane equation is $\mathbf{X}\bullet\mathbf{n} + \frac{\mathbf{W}}{|\mathbf{W}|} = 0$.) The distance from the hyperplane to the origin is $\frac{w_{n+1}}{|\mathbf{W}|}$, and the distance from an arbitrary point, $\mathbf{X}$, to the hyperplane is $\frac{\mathbf{X}\bullet\mathbf{W} + w_{n+1}}{|\mathbf{W}|}$. When the distance from the hyperplane to the origin is negative (that is, when $w_{n+1} < 0$), then the origin is on the negative side of the hyperplane (that is, the side for which $\mathbf{X}\bullet\mathbf{W} + w_{n+1} < 0$).

Adjusting the weight vector, $\mathbf{W}$, changes the orientation of the hyperplane; adjusting w_{n+1} changes the position of the hyperplane (relative to the origin). Thus, training of a TLU can be achieved by adjusting the values of the weights. In this way the hyperplane can be moved so that the TLU implements different (linearly separable) functions of the input.

4.1.2 Special Cases of Linearly Separable Functions

Terms

Any term of size k can be implemented by a TLU with a weight from each of those inputs corresponding to variables occurring in the term. A weight of $+1$ is used from an input corresponding to a positive literal, and a weight of -1 is used from an input corresponding to a negative literal. (Literals not mentioned in the term have weights of zero—that is, no connection at all—from their inputs.) The threshold, θ, is set equal to $k_p - 1/2$, where k_p is the number of positive literals in the term. Such a TLU implements a hyperplane boundary that is parallel to a subface of dimension $(n - k)$ of the unit hypercube. We show a three-dimensional example in Fig. 4.3. Thus, linearly separable functions are a superset of terms.

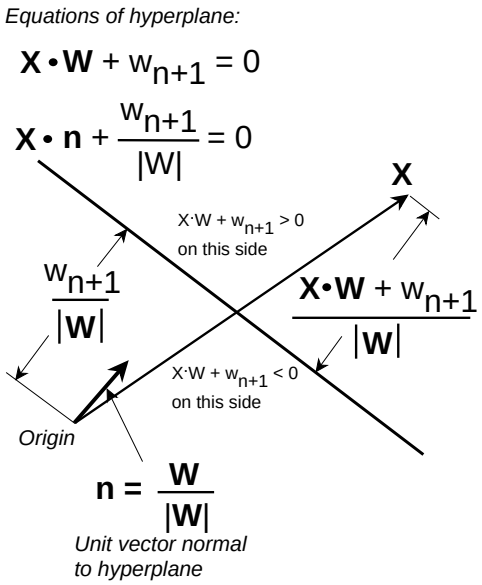


Figure 4.2: TLU Geometry

Clauses

The negation of a clause is a term. For example, the negation of the clause $f = x_1 + x_2 + x_3$ is the term $\overline{f} = \overline{x_1}\ \overline{x_2}\ \overline{x_3}$. A hyperplane can be used to implement this term. If we "invert" the hyperplane, it will implement the clause instead. Inverting a hyperplane is done by multiplying all of the TLU weights—even w_{n+1}—by -1. This process simply changes the orientation of the hyperplane—flipping it around by 180 degrees and thus changing its "positive side." Therefore, linearly separable functions are also a superset of clauses. We show an example in Fig. 4.4.

4.1.3 Error-Correction Training of a TLU

There are several procedures that have been proposed for adjusting the weights of a TLU. We present next a family of *incremental* training procedures with parameter c. These methods make adjustments to the weight vector only when the TLU being trained makes an error on a training pattern; they are called *error-correction* procedures. We use *augmented* feature and weight vectors in describing them.

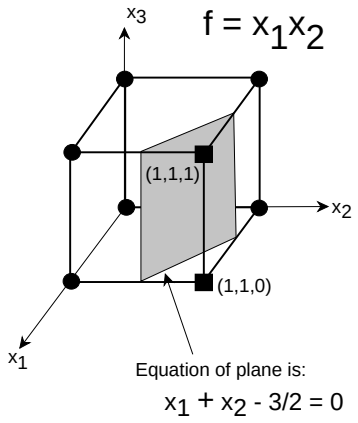


Figure 4.3: Implementing a Term

a. We start with a finite training set, Ξ, of vectors, $\mathbf{Y}_i$, and their binary labels.

b. Compose an infinite training sequence, Σ, of vectors from Ξ and their labels such that each member of Ξ occurs infinitely often in Σ. Set the initial weight values of an TLU to arbitrary values.

c. Repeat forever:

   Present the next vector, $\mathbf{Y}_i$, in Σ to the TLU and note its response.

   (a) If the TLU responds correctly, make no change in the weight vector.

   (b) If $\mathbf{Y}_i$ is supposed to produce an output of 0 and produces an output of 1 instead, modify the weight vector as follows:

$$\mathbf{V} \longleftarrow \mathbf{V} - c_i \mathbf{Y}_i$$

   where c_i is a positive real number called the *learning rate parameter* (whose value is differerent in different instances of this family of procedures and may depend on i).

   Note that after this adjustment the new dot product will be $(\mathbf{V} - c_i \mathbf{Y}_i) \bullet \mathbf{Y}_i = \mathbf{V} \bullet \mathbf{Y}_i - c_i \mathbf{Y}_i \bullet \mathbf{Y}_i$, which is smaller than it was before the weight adjustment.

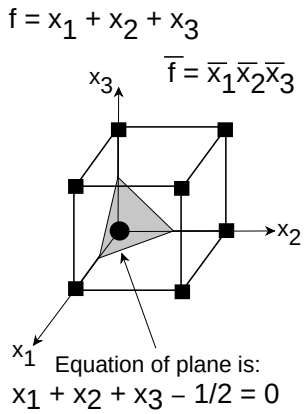


Figure 4.4: Implementing a Clause

(c) If $\mathbf{Y}_i$ is supposed to produce an output of 1 and produces an output of 0 instead, modify the weight vector as follows:

$$\mathbf{V} \longleftarrow \mathbf{V} + c_i \mathbf{Y}_i$$

In this case, the new dot product will be $(\mathbf{V} + c_i \mathbf{Y}_i) \bullet \mathbf{Y}_i = \mathbf{V} \bullet \mathbf{Y}_i + c_i \mathbf{Y}_i \bullet \mathbf{Y}_i$, which is larger than it was before the weight adjustment.

Note that all three of these cases can be combined in the following rule:

$$\mathbf{V} \longleftarrow \mathbf{V} + c_i (d_i - f_i) \mathbf{Y}_i$$

where d_i is the desired response (1 or 0) for $\mathbf{Y}_i$, and f_i is the actual response (1 or 0) for $\mathbf{Y}_i$.]

Note also that because the weight vector $\mathbf{V}$ now includes the w_{n+1} threshold component, the threshold of the TLU is also changed by these adjustments.

We identify two versions of this procedure:

1) In the *fixed-increment procedure*, the learning rate parameter, c_i, is the same fixed, positive constant for all i. Depending on the value of this constant, the weight adjustment may or may not correct the response to an erroneously classified feature vector.

2) In the *fractional-correction procedure*, the parameter c_i is set to $\lambda \frac{\mathbf{Y_i} \bullet \mathbf{V}}{\mathbf{Y_i} \bullet \mathbf{Y_i}}$, where $\mathbf{V}$ is the weight vector *before* it is changed. Note that if $\lambda = 0$, no correction takes place at all. If $\lambda = 1$, the correction is just sufficient to make $\mathbf{Y_i} \bullet \mathbf{V} = 0$. If $\lambda > 1$, the error will be corrected.

It can be proved that if there is some weight vector, $\mathbf{V}$, that produces a correct output for all of the feature vectors in Ξ, then after a finite number of feature vector presentations, the fixed-increment procedure will find such a weight vector and thus make no more weight changes. The same result holds for the fractional-correction procedure if $1 < \lambda \leq 2$.

For additional background, proofs, and examples of error-correction procedures, see [Nilsson, 1990].

See [Maass & Turán, 1994] for a hyperplane-finding procedure that makes no more than $O(n^2 \log n)$ mistakes.

4.1.4 Weight Space

We can give an intuitive idea about how these procedures work by considering what happens to the augmented weight vector in "weight space" as corrections are made. We use augmented vectors in our discussion here so that the threshold function compares the dot product, $\mathbf{Y}_i \bullet \mathbf{V}$, against a threshold of 0. A particular weight vector, $\mathbf{V}$, then corresponds to a point in $(n+1)$-dimensional weight space. Now, for any pattern vector, $\mathbf{Y}_i$, consider the locus of all points in weight space corresponding to weight vectors yielding $\mathbf{Y}_i \bullet \mathbf{V} = 0$. This locus is a hyperplane passing through the origin of the $(n+1)$-dimensional space. Each pattern vector will have such a hyperplane corresponding to it. Weight points in one of the half-spaces defined by this hyperplane will cause the corresponding pattern to yield a dot product less than 0, and weight points in the other half-space will cause the corresponding pattern to yield a dot product greater than 0.

We show a schematic representation of such a weight space in Fig. 4.5. There are four pattern hyperplanes, 1, 2, 3, 4 , corresponding to patterns $\mathbf{Y}_1$, $\mathbf{Y}_2$, $\mathbf{Y}_3$, $\mathbf{Y}_4$, respectively, and we indicate by an arrow the half-space for each in which weight vectors give dot products greater than 0. Suppose we wanted weight values that would give positive responses for patterns $\mathbf{Y}_1$, $\mathbf{Y}_3$, and $\mathbf{Y}_4$, and a negative response for pattern $\mathbf{Y}_2$. The weight point, $\mathbf{V}$, indicated in the figure is one such set of weight values.

The question of whether or not there exists a weight vector that gives desired responses for a given set of patterns can be given a geometric interpretation. To do so involves reversing the "polarity" of those hyperplanes corresponding to patterns for which a negative response is desired. If we do that for our example above, we get the weight space diagram shown in Fig. 4.6.

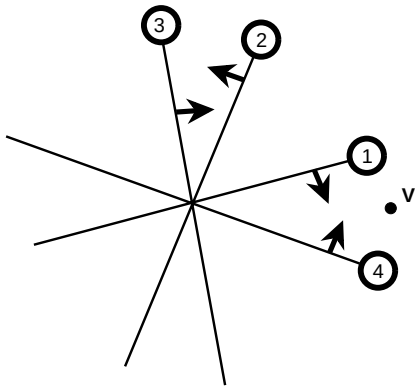


Figure 4.5: Weight Space

If a weight vector exists that correctly classifies a set of patterns, then the half-spaces defined by the correct responses for these patterns will have a non-empty intersection, called the solution region. The solution region will be a "hyper-wedge" region whose vertex is at the origin of weight space and whose cross-section increases with increasing distance from the origin. This region is shown shaded in Fig. 4.6. (The boxed numbers show, for later purposes, the number of errors made by weight vectors in each of the regions.) The fixed-increment error-correction procedure changes a weight vector by moving it normal to any pattern hyperplane for which that weight vector gives an incorrect response. Suppose in our example that we present the patterns in the sequence $\mathbf{Y}_1$, $\mathbf{Y}_2$, $\mathbf{Y}_3$, $\mathbf{Y}_4$, and start the process with a weight point $\mathbf{V}_1$, as shown in Fig. 4.7. Starting at $\mathbf{V}_1$, we see that it gives an incorrect response for pattern $\mathbf{Y}_1$, so we move $\mathbf{V}_1$ to $\mathbf{V}_2$ in a direction normal to plane 1. (That is what adding $\mathbf{Y}_1$ to $\mathbf{V}_1$ does.) $\mathbf{Y}_2$ gives an incorrect response for pattern $\mathbf{Y}_2$, and so on. Ultimately, the responses are only incorrect for planes bounding the solution region. Some of the subsequent corrections may overshoot the solution region, but eventually we work our way out far enough in the solution region that corrections (for a fixed increment size) take us within it. The proofs for convergence of the fixed-increment rule make this intuitive argument precise.

4.1.5 The Widrow-Hoff Procedure

The Widrow-Hoff procedure (also called the *LMS* or the *delta* procedure) attempts to find weights that minimize a squared-error function between the

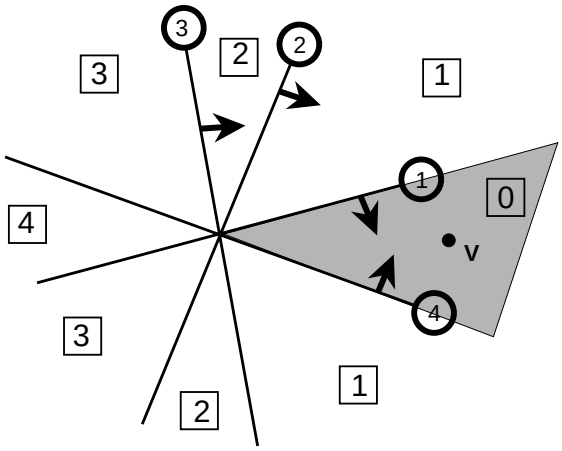


Figure 4.6: Solution Region in Weight Space

pattern labels and the dot product computed by a TLU. For this purpose, the pattern labels are assumed to be either $+1$ or -1 (instead of 1 or 0). The squared error for a pattern, $\mathbf{X}_i$, with label d_i (for desired output) is:

$$\varepsilon_i = (d_i - \sum_{j=1}^{n+1} x_{ij} w_j)^2$$

where x_{ij} is the j-th component of $\mathbf{X}_i$. The total squared error (over all patterns in a training set, Ξ, containing m patterns) is then:

$$\varepsilon = \sum_{i=1}^{m} (d_i - \sum_{j=1}^{n+1} x_{ij} w_j)^2$$

We want to choose the weights w_j to minimize this squared error. One way to find such a set of weights is to start with an arbitrary weight vector and move it along the negative gradient of ε as a function of the weights. Since ε is quadratic in the w_j, we know that it has a global minimum, and thus this *steepest descent* procedure is guaranteed to find the minimum. Each component of the gradient is the partial derivative of ε with respect to one of the weights. One problem with taking the partial derivative of ε is that ε depends on *all* the input vectors in Ξ. Often, it is preferable to use an incremental procedure in which we try the TLU on just one element, $\mathbf{X}_i$,

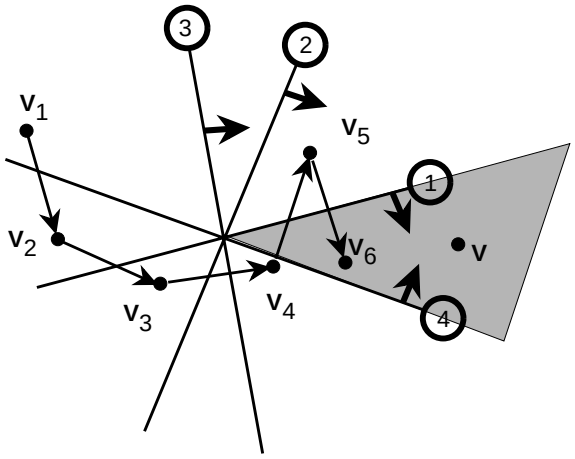


Figure 4.7: Moving Into the Solution Region

of Ξ at a time, compute the gradient of the single-pattern squared error, ε_i, make the appropriate adjustment to the weights, and then try another member of Ξ. Of course, the results of the incremental version can only approximate those of the batch one, but the approximation is usually quite effective. We will be describing the incremental version here.

The j-th component of the gradient of the single-pattern error is:

$$\frac{\partial \varepsilon_i}{\partial w_j} = -2(d_i - \sum_{j=1}^{n+1} x_{ij} w_j) x_{ij}$$

An adjustment in the direction of the negative gradient would then change each weight as follows:

$$w_j \longleftarrow w_j + c_i(d_i - f_i)x_{ij}$$

where $f_i = \sum_{j=1}^{n+1} x_{ij} w_j$, and c_i governs the size of the adjustment. The entire weight vector (in augmented, or $\mathbf{V}$, notation) is thus adjusted according to the following rule:

$$\mathbf{V} \longleftarrow \mathbf{V} + c_i(d_i - f_i)\mathbf{Y}_i$$

where, as before, $\mathbf{Y}_i$ is the i-th augmented pattern vector.

The Widrow-Hoff procedure makes adjustments to the weight vector whenever the dot product itself, $\mathbf{Y}_i \bullet \mathbf{V}$, does not equal the specified desired target value, d_i (which is either 1 or -1). The learning-rate factor, c_i, might decrease with time toward 0 to achieve asymptotic convergence. The Widrow-Hoff formula for changing the weight vector has the same form as the standard fixed-increment error-correction formula. The only difference is that f_i is the thresholded response of the TLU in the error-correction case while it is the dot product itself for the Widrow-Hoff procedure.

Finding weight values that give the desired dot products corresponds to solving a set of linear equalities, and the Widrow-Hoff procedure can be interpreted as a descent procedure that attempts to minimize the mean-squared-error between the actual and desired values of the dot product. (For more on Widrow-Hoff and other related procedures, see [Duda & Hart, 1973, pp. 151ff].)

Examples of training curves for TLU's; performance on training set; performance on test set; cumulative number of corrections.

4.1.6 Training a TLU on Non-Linearly-Separable Training Sets

When the training set is not linearly separable (perhaps because of noise or perhaps inherently), it may still be desired to find a "best" separating hyperplane. Typically, the error-correction procedures will not do well on non-linearly-separable training sets because they will continue to attempt to correct inevitable errors, and the hyperplane will never settle into an acceptable place.

Several methods have been proposed to deal with this case. First, we might use the Widrow-Hoff procedure, which (although it will not converge to zero error on non-linearly separable problems) will give us a weight vector that minimizes the mean-squared-error. A mean-squared-error criterion often gives unsatisfactory results, however, because it prefers many small errors to a few large ones. As an alternative, error correction with a continuous decrease toward zero of the value of the learning rate constant, c, will result in ever decreasing changes to the hyperplane. Duda [Duda, 1966] has suggested keeping track of the average value of the weight vector during error correction and using this average to give a separating hyperplane that performs reasonably well on non-linearly-separable problems. Gallant [Gallant, 1986] proposed what he called the "pocket algorithm." As described in [Hertz, Krogh, & Palmer, 1991, p. 160]:

> . . . the pocket algorithm . . . consists simply in storing (or "putting in your pocket") the set of weights which has had the longest unmodified run of successes so far. The algorithm is stopped after some chosen time t . . .

After stopping, the weights in the pocket are used as a set that should give a small number of errors on the training set. Error-correction proceeds as usual with the ordinary set of weights.

Also see methods proposed by [John, 1995] and by [Marchand & Golea, 1993]. The latter is claimed to outperform the pocket algorithm.

4.2 Linear Machines

The natural generalization of a (two-category) TLU to an R-category classifier is the structure, shown in Fig. 4.8, called a *linear machine*. Here, to use more familiar notation, the $\mathbf{W}$s and $\mathbf{X}$ are meant to be augmented vectors (with an (n+1)-st component). Such a structure is also sometimes called a "competitive" net or a "winner-take-all" net. The output of the linear machine is one of the numbers, $\{1, \ldots, R\}$, corresponding to which dot product is largest. Note that when $R = 2$, the linear machine reduces to a TLU with weight vector $\mathbf{W} = (\mathbf{W}_1 - \mathbf{W}_2)$.

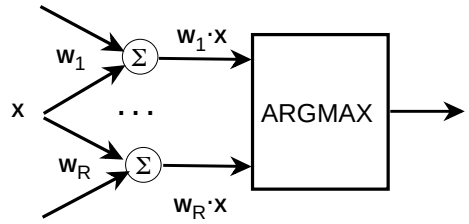


Figure 4.8: A Linear Machine

The diagram in Fig. 4.9 shows the character of the regions in a 2-dimensional space created by a linear machine for $R = 5$. In n dimensions, every pair of regions is either separated by a section of a hyperplane or is non-adjacent.

To train a linear machine, there is a straightforward generalization of the 2-category error-correction rule. Assemble the patterns in the training set into a sequence as before.

a. If the machine classifies a pattern correctly, no change is made to any of the weight vectors.

b. If the machine mistakenly classifies a category u pattern, $\mathbf{X}_i$, in category v ($u \neq v$), then:

$$\mathbf{W}_u \longleftarrow \mathbf{W}_u + c_i \mathbf{X}_i$$

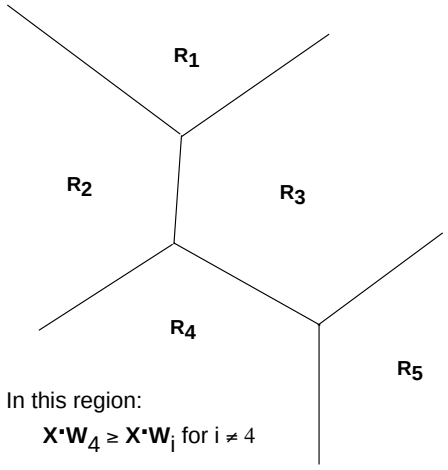


Figure 4.9: Regions For a Linear Machine

and

$$\mathbf{W}_v \longleftarrow \mathbf{W}_v - c_i \mathbf{X}_i$$

and all other weight vectors are not changed.

This correction increases the value of the u-th dot product and decreases the value of the v-th dot product. Just as in the 2-category fixed increment procedure, this procedure is guaranteed to terminate, for constant c_i, if there exists weight vectors that make correct separations of the training set. Note that when $R = 2$, this procedure reduces to the ordinary TLU error-correction procedure. A proof that this procedure terminates is given in [Nilsson, 1990, pp. 88-90] and in [Duda & Hart, 1973, pp. 174-177].

4.3 Networks of TLUs

4.3.1 Motivation and Examples

Layered Networks

To classify correctly all of the patterns in non-linearly-separable training sets requires separating surfaces more complex than hyperplanes. One way

to achieve more complex surfaces is with networks of TLUs. Consider, for example, the 2-dimensional, even parity function, $f = x_1 x_2 + \overline{x_1}\ \overline{x_2}$. No single line through the 2-dimensional square can separate the vertices (1,1) and (0,0) from the vertices (1,0) and (0,1)—the function is not linearly separable and thus cannot be implemented by a single TLU. But, the network of three TLUs shown in Fig. 4.10 does implement this function. In the figure, we show the weight values along input lines to each TLU and the threshold value inside the circle representing the TLU.

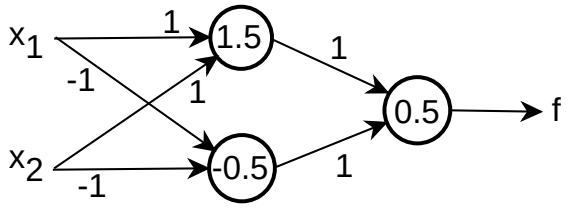


Figure 4.10: A Network for the Even Parity Function

The function implemented by a network of TLUs depends on its topology as well as on the weights of the individual TLUs. *Feedforward* networks have no cycles; in a feedforward network no TLU's input depends (through zero or more intermediate TLUs) on that TLU's output. (Networks that are not feedforward are called *recurrent* networks). If the TLUs of a feedforward network are arranged in layers, with the elements of layer j receiving inputs only from TLUs in layer $j-1$, then we say that the network is a *layered, feedforward network*. The network shown in Fig. 4.10 is a layered, feedforward network having two layers (of weights). (Some people count the layers of TLUs and include the inputs as a layer also; they would call this network a three-layer network.) In general, a feedforward, layered network has the structure shown in Fig. 4.11. All of the TLUs except the "output" units are called hidden units (they are "hidden" from the output).

Implementing DNF Functions by Two-Layer Networks

We have already defined k-term DNF functions—they are DNF functions having k terms. A k-term DNF function can be implemented by a two-layer network with k units in the hidden layer—to implement the k terms—and one output unit to implement the disjunction of these terms. Since any Boolean function has a DNF form, any Boolean function can be implemented by some two-layer network of TLUs. As an example, consider the

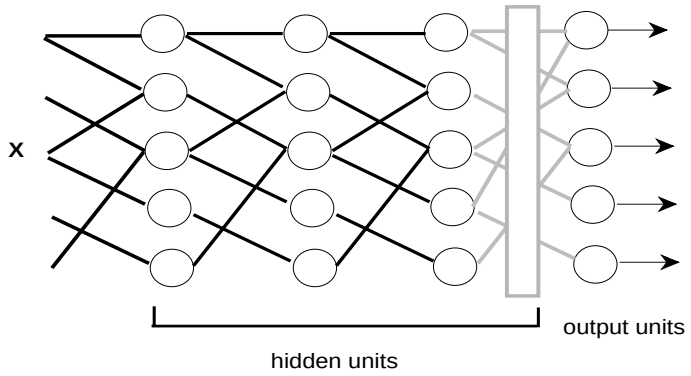


Figure 4.11: A Layered, Feedforward Network

function $f = x_1x_2 + x_2\overline{x_3} + x_1\overline{x_3}$. The form of the network that implements this function is shown in Fig. 4.12. (We leave it to the reader to calculate appropriate values of weights and thresholds.) The 3-cube representation of the function is shown in Fig. 4.13. The network of Fig. 4.12 can be designed so that each hidden unit implements one of the planar boundaries shown in Fig. 4.13.

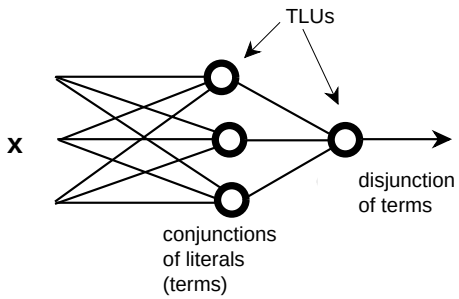


Figure 4.12: A Two-Layer Network

To train a two-layer network that implements a k-term DNF function, we first note that the output unit implements a disjunction, so the weights in the final layer are fixed. The weights in the first layer (except for the "threshold weights") can all have values of 1, -1, or 0. Later, we will

$$f = x_1x_2 + x_2\overline{x_3} + x_1\overline{x_3}$$

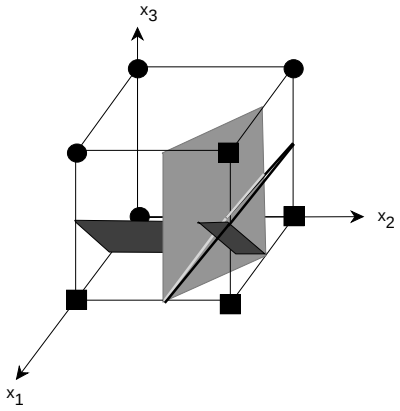


Figure 4.13: Three Planes Implemented by the Hidden Units

Discuss half-space intersections, half-space unions, NP-hardness of optimal versions, single-side-error-hypeplane methods, relation to "AQ" methods.

present a training procedure for this first layer of weights.

Important Comment About Layered Networks

Adding additional layers cannot compensate for an inadequate first layer of TLUs. The first layer of TLUs partitions the feature space so that no two differently labeled vectors are in the same region (that is, so that no two such vectors yield the same set of outputs of the first-layer units). If the first layer does not partition the feature space in this way, then regardless of what subsequent layers do, the final outputs will not be consistent with the labeled training set.

Add diagrams showing the non-linear transformation performed by a layered network.

4.3.2 Madalines

Two-Category Networks

An interesting example of a layered, feedforward network is the two-layer one which has an odd number of hidden units, and a "vote-taking" TLU as the output unit. Such a network was called a "Madaline" (for many adalines by Widrow. Typically, the response of the vote taking unit is defined to be the response of the majority of the hidden units, although

other output logics are possible. Ridgway [Ridgway, 1962] proposed the following error-correction rule for adjusting the weights of the hidden units of a Madaline:

- If the Madaline correctly classifies a pattern, $\mathbf{X}_i$, no corrections are made to any of the hidden units' weight vectors,
- If the Madaline incorrectly classifies a pattern, $\mathbf{X}_i$, then determine the minimum number of hidden units whose responses need to be changed (from 0 to 1 or from 1 to 0—depending on the type of error) in order that the Madaline would correctly classify $\mathbf{X}_i$. Suppose that minimum number is k_i. Of those hidden units voting incorrectly, change the weight vectors of those k_i of them whose dot products are closest to 0 by using the error correction rule:

$$\mathbf{W} \longleftarrow \mathbf{W} + c_i(d_i - f_i)\mathbf{X}_i$$

  where d_i is the desired response of the hidden unit (0 or 1) and f_i is the actual response (0 or 1). (We assume augmented vectors here even though we are using $\mathbf{X}$, $\mathbf{W}$ notation.)

That is, we perform error-correction on just enough hidden units to correct the vote to a majority voting correctly, and we change those that are easiest to change. There are example problems in which even though a set of weight values exists for a given Madaline structure such that it could classify all members of a training set correctly, this procedure will fail to find them. Nevertheless, the procedure works effectively in most experiments with it.

We leave it to the reader to think about how this training procedure could be modified if the output TLU implemented an *or* function (or an *and* function).

R-Category Madalines and Error-Correcting Output Codes

If there are k hidden units ($k > 1$) in a two-layer network, their responses correspond to vertices of a k-dimensional hypercube. The ordinary two-category Madaline identifies two special points in this space, namely the vertex consisting of k 1's and the vertex consisting of k 0's. The Madaline's response is 1 if the point in "hidden-unit-space" is closer to the all 1's vertex than it is to the all 0's vertex. We could design an R-category Madaline by identifying R vertices in hidden-unit space and then classifying a pattern

according to which of these vertices the hidden-unit response is closest to. A machine using that idea was implemented in the early 1960s at SRI [Brain, *et al.*, 1962]. It used the fact that the 2^p so-called *maximal-length shift-register sequences* [Peterson, 1961, pp. 147ff] in a $(2^p - 1)$-dimensional Boolean space are mutually equidistant (for any integer p). For similar, more recent work see [Dietterich & Bakiri, 1991].

4.3.3 Piecewise Linear Machines

A two-category training set is linearly separable if there exists a threshold function that correctly classifies all members of the training set. Similarly, we can say that an R-category training set is linearly separable if there exists a linear machine that correctly classifies all members of the training set. When an R-category problem is not linearly separable, we need a more powerful classifier. A candidate is a structure called a *piecewise linear (PWL)* machine illustrated in Fig. 4.14.

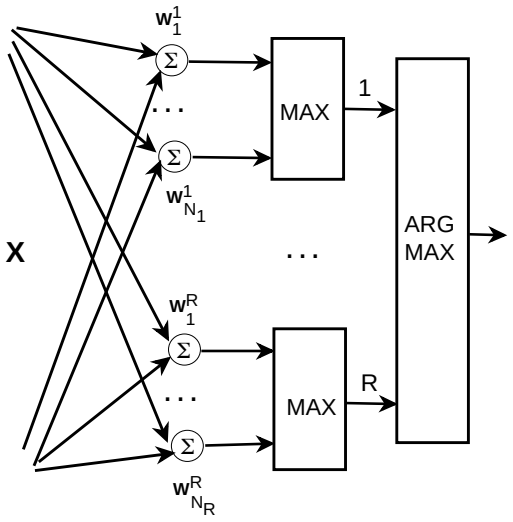


Figure 4.14: A Piecewise Linear Machine

The PWL machine groups its weighted summing units into R banks corresponding to the R categories. An input vector $\mathbf{X}$ is assigned to that category corresponding to the bank with the largest weighted sum. We can

use an error-correction training algorithm similar to that used for a linear machine. If a pattern is classified incorrectly, we subtract (a constant times) the pattern vector from the weight vector producing the largest dot product (it was incorrectly the largest) and add (a constant times) the pattern vector to that weight vector in the correct bank of weight vectors whose dot product is locally largest in that bank. (Again, we use augmented vectors here.) Unfortunately, there are example training sets that are separable by a given PWL machine structure but for which this error-correction training method fails to find a solution. The method does appear to work well in some situations [Duda & Fossum, 1966], although [Nilsson, 1965, page 89] observed that "it is probably not a very effective method for training PWL machines having more than three [weight vectors] in each bank."

4.3.4 Cascade Networks

Another interesting class of feedforward networks is that in which all of the TLUs are ordered and each TLU receives inputs from all of the pattern components and from all TLUs lower in the ordering. Such a network is called a *cascade* network. An example is shown in Fig. 4.15 in which the TLUs are labeled by the linearly separable functions (of their inputs) that they implement. Each TLU in the network implements a set of 2^k parallel hyperplanes, where k is the number of TLUs from which it receives inputs. (Each of the k preceding TLUs can have an output of 1 or 0; that's 2^k different combinations—resulting in 2^k different positions for the parallel hyperplanes.) We show a 3-dimensional sketch for a network of two TLUs in Fig. 4.16. The reader might consider how the n-dimensional parity function might be implemented by a cascade network having $\log_2 n$ TLUs.

Cascade networks might be trained by first training L_1 to do as good a job as possible at separating all the training patterns (perhaps by using the pocket algorithm, for example), then training L_2 (including the weight from L_1 to L_2) also to do as good a job as possible at separating all the training patterns, and so on until the resulting network classifies the patterns in the training set satisfactorily.

Also mention the "cascade-correlation" method of [Fahlman & Lebiere, 1990].

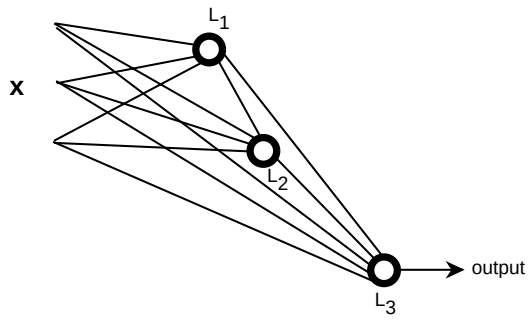


Figure 4.15: A Cascade Network

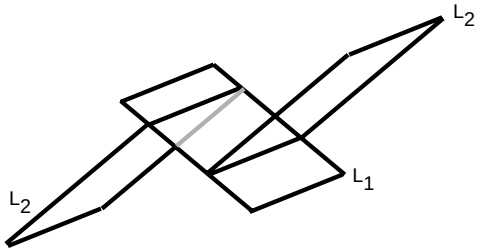


Figure 4.16: Planes Implemented by a Cascade Network with Two TLUs

4.4 Training Feedforward Networks by Backpropagation

4.4.1 Notation

The general problem of training a network of TLUs is difficult. Consider, for example, the layered, feedforward network of Fig. 4.11. If such a network makes an error on a pattern, there are usually several different ways in which the error can be corrected. It is difficult to assign "blame" for the error to any particular TLU in the network. Intuitively, one looks for weight-adjusting procedures that move the network in the correct direction (relative to the error) by making minimal changes. In this spirit, the Widrow-Hoff method of gradient descent has been generalized to deal with multilayer networks.

In explaining this generalization, we use Fig. 4.17 to introduce some notation. This network has only one output unit, but, of course, it is possible to have several TLUs in the output layer—each implementing a different function. Each of the layers of TLUs will have outputs that we take to be the components of vectors, just as the input features are components of an input vector. The j-th layer of TLUs ($1 \leq j < k$) will have as their outputs the vector $\mathbf{X}^{(j)}$. The input feature vector is denoted by $\mathbf{X}^{(0)}$, and the final output (of the k-th layer TLU) is f. Each TLU in each layer has a weight vector (connecting it to its inputs) and a threshold; the i-th TLU in the j-th layer has a weight vector denoted by $\mathbf{W}_i^{(j)}$. (We will assume that the "threshold weight" is the last component of the associated weight vector; we might have used $\mathbf{V}$ notation instead to include this threshold component, but we have chosen here to use the familiar $\mathbf{X}, \mathbf{W}$ notation, assuming that these vectors are "augmented" as appropriate.) We denote the weighted sum input to the i-th threshold unit in the j-th layer by $s_i^{(j)}$. (That is, $s_i^{(j)} = \mathbf{X}^{(j-1)} \bullet \mathbf{W}_i^{(j)}$.) The number of TLUs in the j-th layer is given by m_j. The vector $\mathbf{W}_i^{(j)}$ has components $w_{l,i}^{(j)}$ for $l = 1, \ldots, m_{(j-1)} + 1$.

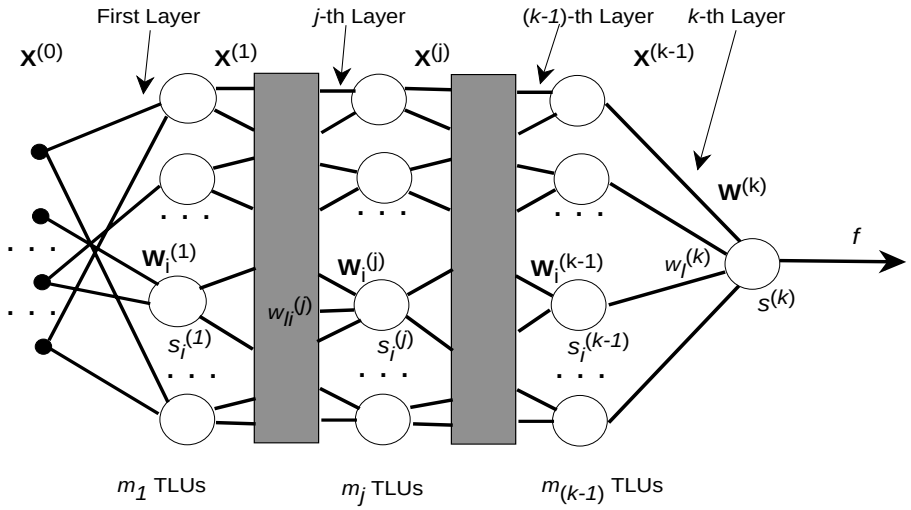


Figure 4.17: A k-layer Network

4.4.2 The Backpropagation Method

A gradient descent method, similar to that used in the Widrow Hoff method, has been proposed by various authors for training a multi-layer, feedforward network. As before, we define an error function on the final output of the network and we adjust each weight in the network so as to minimize the error. If we have a desired response, d_i, for the i-th input vector, $\mathbf{X}_i$, in the training set, Ξ, we can compute the squared error over the entire training set to be:

$$\varepsilon = \sum_{\mathbf{X}_i \ \epsilon \ \Xi} (d_i - f_i)^2$$

where f_i is the actual response of the network for input $\mathbf{X}_i$. To do gradient descent on this squared error, we adjust each weight in the network by an amount proportional to the negative of the partial derivative of ε with respect to that weight. Again, we use a single-pattern error function so that we can use an incremental weight adjustment procedure. The squared error for a single input vector, $\mathbf{X}$, evoking an output of f when the desired output is d is:

$$\varepsilon = (d - f)^2$$

It is convenient to take the partial derivatives of ε with respect to the various weights in groups corresponding to the weight vectors. We define a partial derivative of a quantity ϕ, say, with respect to a weight vector, $\mathbf{W}_i^{(j)}$, thus:

$$\frac{\partial \phi}{\partial \mathbf{W}_i^{(j)}} \stackrel{\text{def}}{=} \left[\frac{\partial \phi}{\partial w_{1i}^{(j)}}, \ldots, \frac{\partial \phi}{\partial w_{li}^{(j)}}, \ldots, \frac{\partial \phi}{\partial w_{m_{j-1}+1,i}^{(j)}} \right]$$

where $w_{li}^{(j)}$ is the l-th component of $\mathbf{W}_i^{(j)}$. This vector partial derivative of ϕ is called the *gradient* of ϕ with respect to $\mathbf{W}$ and is sometimes denoted by $\nabla_{\mathbf{W}} \phi$.

Since ε's dependence on $\mathbf{W}_i^{(j)}$ is entirely through $s_i^{(j)}$, we can use the chain rule to write:

$$\frac{\partial \varepsilon}{\partial \mathbf{W}_i^{(j)}} = \frac{\partial \varepsilon}{\partial s_i^{(j)}} \frac{\partial s_i^{(j)}}{\partial \mathbf{W}_i^{(j)}}$$

Because $s_i^{(j)} = \mathbf{X}^{(j-1)} \bullet \mathbf{W}_i^{(j)}$, $\frac{\partial s_i^{(j)}}{\partial \mathbf{W}_i^{(j)}} = \mathbf{X}^{(j-1)}$. Substituting yields:

$$\frac{\partial \varepsilon}{\partial \mathbf{W}_i^{(j)}} = \frac{\partial \varepsilon}{\partial s_i^{(j)}} \mathbf{X}^{(j-1)}$$

Note that $\frac{\partial \varepsilon}{\partial s_i^{(j)}} = -2(d-f)\frac{\partial f}{\partial s_i^{(j)}}$. Thus,

$$\frac{\partial \varepsilon}{\partial \mathbf{W}_i^{(j)}} = -2(d-f)\frac{\partial f}{\partial s_i^{(j)}} \mathbf{X}^{(j-1)}$$

The quantity $(d-f)\frac{\partial f}{\partial s_i^{(j)}}$ plays an important role in our calculations; we shall denote it by $\delta_i^{(j)}$. Each of the $\delta_i^{(j)}$'s tells us how sensitive the squared error of the network output is to changes in the input to each threshold function. Since we will be changing weight vectors in directions along their negative gradient, our fundamental rule for weight changes throughout the network will be:

$$\mathbf{W}_i^{(j)} \leftarrow \mathbf{W}_i^{(j)} + c_i^{(j)} \delta_i^{(j)} \mathbf{X}^{(j-1)}$$

where $c_i^{(j)}$ is the learning rate constant for this weight vector. (Usually, the learning rate constants for all weight vectors in the network are the same.) We see that this rule is quite similar to that used in the error correction procedure for a single TLU. A weight vector is changed by the addition of a constant times its vector of (unweighted) inputs.

Now, we must turn our attention to the calculation of the $\delta_i^{(j)}$'s. Using the definition, we have:

$$\delta_i^{(j)} = (d-f)\frac{\partial f}{\partial s_i^{(j)}}$$

We have a problem, however, in attempting to carry out the partial derivatives of f with respect to the s's. The network output, f, is not continuously differentiable with respect to the s's because of the presence of the threshold functions. Most small changes in these sums do not change f at all, and when f does change, it changes abruptly from 1 to 0 or vice versa.

A way around this difficulty was proposed by Werbos [Werbos, 1974] and (perhaps independently) pursued by several other researchers, for example [Rumelhart, Hinton, & Williams, 1986]. The trick involves replacing

all the threshold functions by differentiable functions called *sigmoids*.[1] The output of a sigmoid function, superimposed on that of a threshold function, is shown in Fig. 4.18. Usually, the sigmoid function used is $f(s) = \frac{1}{1+e^{-s}}$, where s is the input and f is the output.

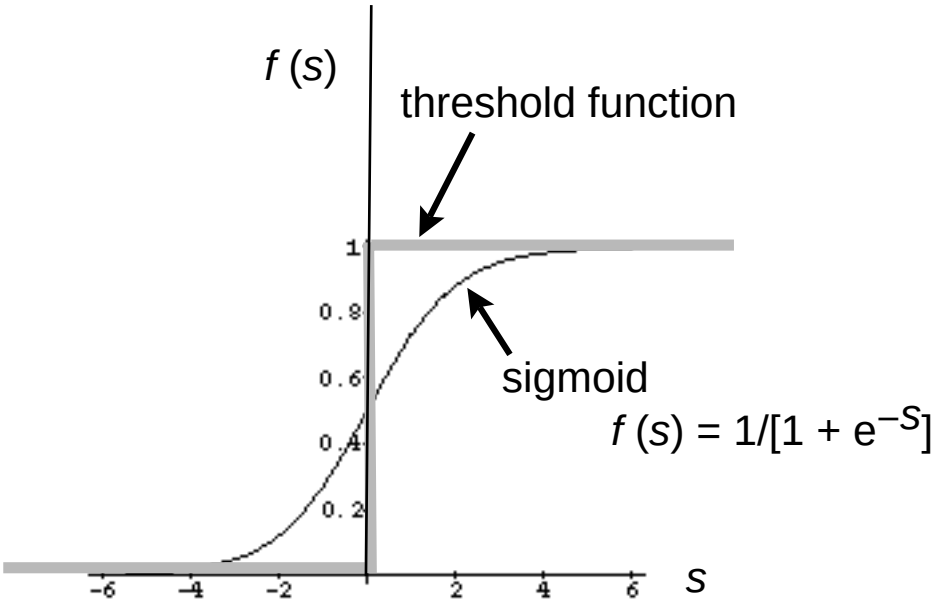


Figure 4.18: A Sigmoid Function

We show the network containing sigmoid units in place of TLUs in Fig. 4.19. The output of the i-th sigmoid unit in the j-th layer is denoted by $f_i^{(j)}$. (That is, $f_i^{(j)} = \frac{1}{1+e^{-s_i^{(j)}}}$.)

4.4.3 Computing Weight Changes in the Final Layer

We first calculate $\delta^{(k)}$ in order to compute the weight change for the final sigmoid unit:

[1] [Russell & Norvig 1995, page 595] attributes the use of this idea to [Bryson & Ho 1969].

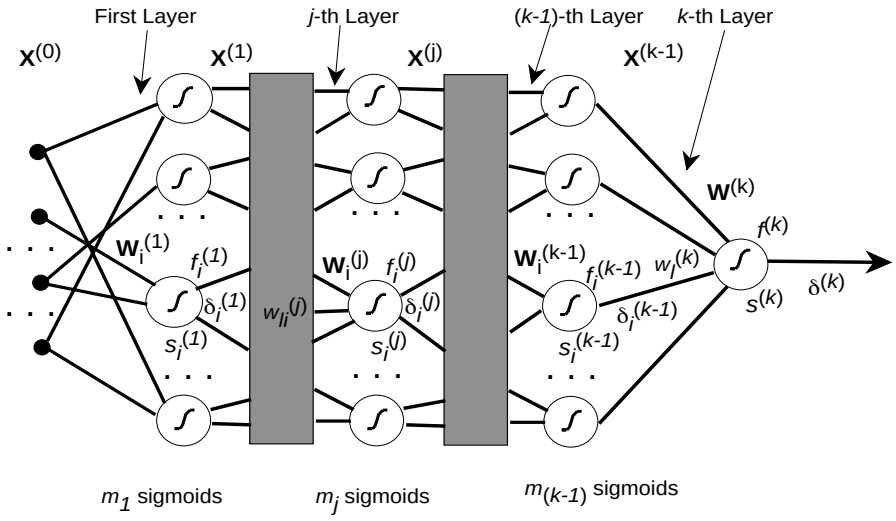


Figure 4.19: A Network with Sigmoid Units

$$\delta^{(k)} = (d - f^{(k)})\frac{\partial f^{(k)}}{\partial s^{(k)}}$$

Given the sigmoid function that we are using, namely $f(s) = \frac{1}{1+e^{-s}}$, we have that $\frac{\partial f}{\partial s} = f(1-f)$. Substituting gives us:

$$\delta^{(k)} = (d - f^{(k)})f^{(k)}(1 - f^{(k)})$$

Rewriting our general rule for weight vector changes, the weight vector in the final layer is changed according to the rule:

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} + c^{(k)}\delta^{(k)}\mathbf{X}^{(k-1)}$$

where $\delta^{(k)} = (d - f^{(k)})f^{(k)}(1 - f^{(k)})$

It is interesting to compare backpropagation to the error-correction rule and to the Widrow-Hoff rule. The backpropagation weight adjustment for the single element in the final layer can be written as:

$$\mathbf{W} \longleftarrow \mathbf{W} + c(d - f)f(1 - f)\mathbf{X}$$

Written in the same format, the error-correction rule is:

$$\mathbf{W} \longleftarrow \mathbf{W} + c(d - f)\mathbf{X}$$

and the Widrow-Hoff rule is:

$$\mathbf{W} \longleftarrow \mathbf{W} + c(d - f)\mathbf{X}$$

The only difference (except for the fact that f is not thresholded in Widrow-Hoff) is the $f(1 - f)$ term due to the presence of the sigmoid function. With the sigmoid function, $f(1 - f)$ can vary in value from 0 to 1. When f is 0, $f(1 - f)$ is also 0; when f is 1, $f(1 - f)$ is 0; $f(1 - f)$ obtains its maximum value of $1/4$ when f is $1/2$ (that is, when the input to the sigmoid is 0). The sigmoid function can be thought of as implementing a "fuzzy" hyperplane. For a pattern far away from this fuzzy hyperplane, $f(1 - f)$ has value close to 0, and the backpropagation rule makes little or no change to the weight values regardless of the desired output. (Small changes in the weights will have little effect on the output for inputs far from the hyperplane.) Weight changes are only made within the region of "fuzz" surrounding the hyperplane, and these changes are in the direction of correcting the error, just as in the error-correction and Widrow-Hoff rules.

4.4.4 Computing Changes to the Weights in Intermediate Layers

Using our expression for the δ's, we can similarly compute how to change each of the weight vectors in the network. Recall:

$$\delta_i^{(j)} = (d - f)\frac{\partial f}{\partial s_i^{(j)}}$$

Again we use a chain rule. The final output, f, depends on $s_i^{(j)}$ through each of the summed inputs to the sigmoids in the $(j+1)$-th layer. So:

$$\delta_i^{(j)} = (d - f)\frac{\partial f}{\partial s_i^{(j)}}$$

$$= (d - f)\left[\frac{\partial f}{\partial s_1^{(j+1)}}\frac{\partial s_1^{(j+1)}}{\partial s_i^{(j)}} + \cdots + \frac{\partial f}{\partial s_l^{(j+1)}}\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}} + \cdots + \frac{\partial f}{\partial s_{m_{j+1}}^{(j+1)}}\frac{\partial s_{m_{j+1}}^{(j+1)}}{\partial s_i^{(j)}}\right]$$

$$= \sum_{l=1}^{m_{j+1}} (d-f) \frac{\partial f}{\partial s_l^{(j+1)}} \frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}} = \sum_{l=1}^{m_{j+1}} \delta_l^{(j+1)} \frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}}$$

It remains to compute the $\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}}$'s. To do that we first write:

$$s_l^{(j+1)} = \mathbf{X}^{(j)} \bullet \mathbf{W}_l^{(j+1)}$$

$$= \sum_{\nu=1}^{m_j+1} f_\nu^{(j)} w_{\nu l}^{(j+1)}$$

And then, since the weights do not depend on the s's:

$$\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}} = \frac{\partial \left[\sum_{\nu=1}^{m_j+1} f_\nu^{(j)} w_{\nu l}^{(j+1)}\right]}{\partial s_i^{(j)}} = \sum_{\nu=1}^{m_j+1} w_{\nu l}^{(j+1)} \frac{\partial f_\nu^{(j)}}{\partial s_i^{(j)}}$$

Now, we note that $\frac{\partial f_\nu^{(j)}}{\partial s_i^{(j)}} = 0$ unless $\nu = i$, in which case $\frac{\partial f_\nu^{(j)}}{\partial s_\nu^{(j)}} = f_\nu^{(j)}(1 - f_\nu^{(j)})$. Therefore:

$$\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}} = w_{il}^{(j+1)} f_i^{(j)} (1 - f_i^{(j)})$$

We use this result in our expression for $\delta_i^{(j)}$ to give:

$$\delta_i^{(j)} = f_i^{(j)} (1 - f_i^{(j)}) \sum_{l=1}^{m_{j+1}} \delta_l^{(j+1)} w_{il}^{(j+1)}$$

The above equation is recursive in the δ's. (It is interesting to note that this expression is independent of the error function; the error function explicitly affects only the computation of $\delta^{(k)}$.) Having computed the $\delta_i^{(j+1)}$'s for layer $j+1$, we can use this equation to compute the $\delta_i^{(j)}$'s. The base case is $\delta^{(k)}$, which we have already computed:

$$\delta^{(k)} = (d - f^{(k)}) f^{(k)} (1 - f^{(k)})$$

We use this expression for the δ's in our generic weight changing rule, namely:

$$\mathbf{W}_i^{(j)} \leftarrow \mathbf{W}_i^{(j)} + c_i^{(j)} \delta_i^{(j)} \mathbf{X}^{(j-1)}$$

Although this rule appears complex, it has an intuitively reasonable explanation. The quantity $\delta^{(k)} = (d - f)f(1 - f)$ controls the overall amount and sign of *all* weight adjustments in the network. (Adjustments diminish as the final output, f, approaches either 0 or 1, because they have vanishing effect on f then.) As the recursion equation for the δ's shows, the adjustments for the weights going *in* to a sigmoid unit in the j-th layer are proportional to the effect that such adjustments have on that sigmoid unit's output (its $f^{(j)}(1 - f^{(j)})$ factor). They are also proportional to a kind of "average" effect that any change in the output of that sigmoid unit will have on the final output. This average effect depends on the weights going *out* of the sigmoid unit in the j-th layer (small weights produce little downstream effect) and the effects that changes in the outputs of $(j+1)$-th layer sigmoid units will have on the final output (as measured by the $\delta^{(j+1)}$'s). These calculations can be simply implemented by "backpropagating" the δ's through the weights in reverse direction (thus, the name *backprop* for this algorithm).

4.4.5 Variations on Backprop

[To be written: problem of local minima, simulated annealing, momemtum (Plaut, *et al.*, 1986, see [Hertz, Krogh, & Palmer, 1991]), quickprop, regularization methods]

Simulated Annealing

To apply simulated annealing, the value of the learning rate constant is gradually decreased with time. If we fall early into an error-function valley that is not very deep (a local minimum), it typically will neither be very broad, and soon a subsequent large correction will jostle us out of it. It is less likely that we will move out of deep valleys, and at the end of the process (with very small values of the learning rate constant), we descend to its deepest point. The process gets its name by analogy with annealing in metallurgy, in which a material's temperature is gradually decreased allowing its crystalline structure to reach a minimal energy state.

4.4.6 An Application: Steering a Van

A neural network system called ALVINN (Autonomous Land Vehicle in a Neural Network) has been trained to steer a Chevy van successfully on ordinary roads and highways at speeds of 55 mph [Pomerleau, 1991, Pomerleau, 1993]. The input to the network is derived from a low-resolution (30 x 32) television image. The TV camera is mounted on the van and looks at the road straight ahead. This image is sampled and produces a stream of 960-dimensional input vectors to the neural network. The network is shown in Fig. 4.20.

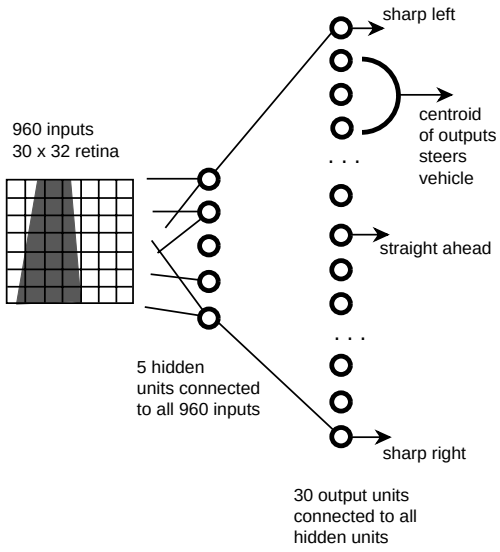


Figure 4.20: The ALVINN Network

The network has five hidden units in its first layer and 30 output units in the second layer; all are sigmoid units. The output units are arranged in a linear order and control the van's steering angle. If a unit near the top of the array of output units has a higher output than most of the other units, the van is steered to the left; if a unit near the bottom of the array has a high output, the van is steered to the right. The "centroid" of the responses of all of the output units is computed, and the van's steering angle is set at a corresponding value between hard left and hard right.

The system is trained by a modified on-line training regime. A driver

drives the van, and his actual steering angles are taken as the correct labels for the corresponding inputs. The network is trained incrementally by backprop to produce the driver-specified steering angles in response to each visual pattern as it occurs in real time while driving.

This simple procedure has been augmented to avoid two potential problems. First, since the driver is usually driving well, the network would never get any experience with far-from-center vehicle positions and/or incorrect vehicle orientations. Also, on long, straight stretches of road, the network would be trained for a long time only to produce straight-ahead steering angles; this training would swamp out earlier training to follow a curved road. We wouldn't want to try to avoid these problems by instructing the driver to drive erratically occasionally, because the system would learn to mimic this erratic behavior.

Instead, each original image is shifted and rotated in software to create 14 additional images in which the vehicle appears to be situated differently relative to the road. Using a model that tells the system what steering angle ought to be used for each of these shifted images, given the driver-specified steering angle for the original image, the system constructs an additional 14 labeled training patterns to add to those encountered during ordinary driver training.

4.5 Synergies Between Neural Network and Knowledge-Based Methods

To be written; discuss rule-generating procedures (such as [Towell & Shavlik, 1992]) and how expert-provided rules can aid neural net training and vice-versa [Towell, Shavlik, & Noordweier, 1990].

4.6 Bibliographical and Historical Remarks

To be added.

Chapter 5

Statistical Learning

5.1 Using Statistical Decision Theory

5.1.1 Background and General Method

Suppose the pattern vector, $\mathbf{X}$, is a random variable whose probability distribution for category 1 is different than it is for category 2. (The treatment given here can easily be generalized to R-category problems.) Specifically, suppose we have the two probability distributions (perhaps probability density functions), $p(\mathbf{X} \mid 1)$ and $p(\mathbf{X} \mid 2)$. Given a pattern, $\mathbf{X}$, we want to use statistical techniques to determine its category—that is, to determine from which distribution it was drawn. These techniques are based on the idea of minimizing the expected value of a quantity similar to the error function we used in deriving the weight-changing rules for backprop.

In developing a decision method, it is necessary to know the relative seriousness of the two kinds of mistakes that might be made. (We might decide that a pattern really in category 1 is in category 2, and vice versa.) We describe this information by a *loss function*, $\lambda(i \mid j)$, for $i, j = 1, 2$. $\lambda(i \mid j)$ represents the loss incurred when we decide a pattern is in category i when really it is in category j. We assume here that $\lambda(1 \mid 1)$ and $\lambda(2 \mid 2)$ are both 0. For any given pattern, $\mathbf{X}$, we want to decide its category in such a way that minimizes the expected value of this loss.

Given a pattern, $\mathbf{X}$, if we decide category i, the expected value of the loss will be:

$$L_{\mathbf{X}}(i) = \lambda(i \mid 1)p(1 \mid \mathbf{X}) + \lambda(i \mid 2)p(2 \mid \mathbf{X})$$

where $p(j \mid \mathbf{X})$ is the probability that given a pattern $\mathbf{X}$, its category is j. Our decision rule will be to decide that $\mathbf{X}$ belongs to category 1 if $L_{\mathbf{X}}(1) \leq L_{\mathbf{X}}(2)$, and to decide on category 2 otherwise.

We can use Bayes' Rule to get expressions for $p(j \mid \mathbf{X})$ in terms of $p(\mathbf{X} \mid j)$, which we assume to be known (or estimatible):

$$p(j \mid \mathbf{X}) = \frac{p(\mathbf{X} \mid j)p(j)}{p(\mathbf{X})}$$

where $p(j)$ is the (a priori) probability of category j (one category may be much more probable than the other); and $p(\mathbf{X})$ is the (a priori) probability of pattern $\mathbf{X}$ being the pattern we are asked to classify. Performing the substitutions given by Bayes' Rule, our decision rule becomes:

Decide category 1 iff:

$$\lambda(1 \mid 1)\frac{p(\mathbf{X} \mid 1)p(1)}{p(\mathbf{X})} + \lambda(1 \mid 2)\frac{p(\mathbf{X} \mid 2)p(2)}{p(\mathbf{X})}$$
$$\leq \lambda(2 \mid 1)\frac{p(\mathbf{X} \mid 1)p(1)}{p(\mathbf{X})} + \lambda(2 \mid 2)\frac{p(\mathbf{X} \mid 2)p(2)}{p(\mathbf{X})}$$

Using the fact that $\lambda(i \mid i) = 0$, and noticing that $p(\mathbf{X})$ is common to both expressions, we obtain,

Decide category 1 iff:

$$\lambda(1 \mid 2)p(\mathbf{X} \mid 2)p(2) \leq \lambda(2 \mid 1)p(\mathbf{X} \mid 1)p(1)$$

If $\lambda(1 \mid 2) = \lambda(2 \mid 1)$ and if $p(1) = p(2)$, then the decision becomes particularly simple:

Decide category 1 iff:

$$p(\mathbf{X} \mid 2) \leq p(\mathbf{X} \mid 1)$$

Since $p(\mathbf{X} \mid j)$ is called the *likelihood* of j with respect to $\mathbf{X}$, this simple decision rule implements what is called a *maximum-likelihood* decision.

More generally, if we define $k(i \mid j)$ as $\lambda(i \mid j)p(j)$, then our decision rule is simply,

Decide category1 iff:

$$k(1 \mid 2)p(\mathbf{X} \mid 2) \leq k(2 \mid 1)p(\mathbf{X} \mid 1)$$

In any case, we need to compare the (perhaps weighted) quantities $p(\mathbf{X} \mid i)$ for $i = 1$ and 2. The exact decision rule depends on the the probability distributions assumed. We will treat two interesting distributions.

5.1.2 Gaussian (or Normal) Distributions

The multivariate (n-dimensional) Gaussian distribution is given by the probability density function:

$$p(\mathbf{X}) = \frac{1}{(2\pi)^{n/2}|\mathbf{\Sigma}|^{1/2}} e^{\frac{-(\mathbf{X}-\mathbf{M})^t \mathbf{\Sigma}^{-1} (\mathbf{X}-\mathbf{M})}{2}}$$

where n is the dimension of the column vector $\mathbf{X}$, the column vector $\mathbf{M}$ is called the *mean vector*, $(\mathbf{X}-\mathbf{M})^t$ is the transpose of the vector $(\mathbf{X}-\mathbf{M})$, $\mathbf{\Sigma}$ is the *covariance matrix* of the distribution (an $n \times n$ symmetric, positive definite matrix), $\mathbf{\Sigma}^{-1}$ is the inverse of the covariance matrix, and $|\mathbf{\Sigma}|$ is the determinant of the covariance matrix.

The mean vector, $\mathbf{M}$, with components $(m_1, \ldots, m_n)$, is the expected value of $\mathbf{X}$ (using this distribution); that is, $\mathbf{M} = E[\mathbf{X}]$. The components of the covariance matrix are given by:

$$\sigma_{ij}^2 = E[(x_i - m_i)(x_j - m_j)]$$

In particular, σ_{ii}^2 is called the *variance* of x_i.

Although the formula appears complex, an intuitive idea for Gaussian distributions can be given when $n = 2$. We show a two-dimensional Gaussian distribution in Fig. 5.1. A three-dimensional plot of the distribution is shown at the top of the figure, and contours of equal probability are shown at the bottom. In this case, the covariance matrix, $\mathbf{\Sigma}$, is such that the

elliptical contours of equal probability are skewed. If the covariance matrix were diagonal, that is if all off-diagonal terms were 0, then the major axes of the elliptical contours would be aligned with the coordinate axes. In general the principal axes are given by the eigenvectors of $\mathbf{\Sigma}$. In any case, the equi-probability contours are all centered on the mean vector, $\mathbf{M}$, which in our figure happens to be at the origin. In general, the formula in the exponent in the Gaussian distribution is a *positive definite quadratic form* (that is, its value is always positive); thus equi-probability contours are hyper-ellipsoids in n-dimensional space.

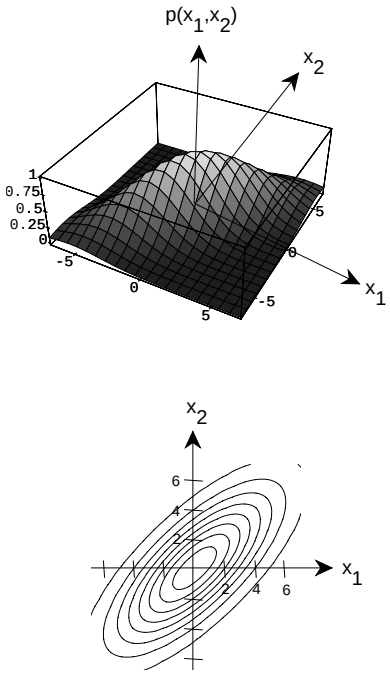


Figure 5.1: The Two-Dimensional Gaussian Distribution

Suppose we now assume that the two classes of pattern vectors that we want to distinguish are each distributed according to a Gaussian distribution but with different means and covariance matrices. That is, one class tends to have patterns clustered around one point in the n-dimensional space, and the other class tends to have patterns clustered around another

point. We show a two-dimensional instance of this problem in Fig. 5.2. (In that figure, we have plotted the sum of the two distributions.) What decision rule should we use to separate patterns into the two appropriate categories?

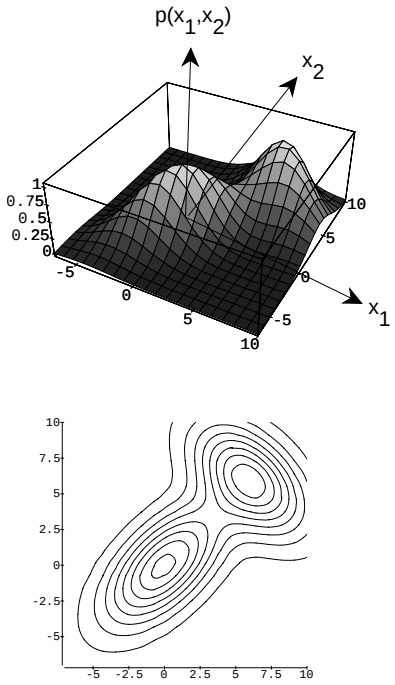


Figure 5.2: The Sum of Two Gaussian Distributions

Substituting the Gaussian distributions into our maximum likelihood formula yields:

Decide category 1 iff:

$$\frac{1}{(2\pi)^{n/2}|\mathbf{\Sigma}_2|^{1/2}} e^{-1/2(\mathbf{X}-\mathbf{M}_2)^t \mathbf{\Sigma}_2^{-1}(\mathbf{X}-\mathbf{M}_2)}$$

is less than or equal to

$$\frac{1}{(2\pi)^{n/2}|\mathbf{\Sigma}_1|^{1/2}} e^{-1/2(\mathbf{X}-\mathbf{M}_1)^t \mathbf{\Sigma}_1^{-1}(\mathbf{X}-\mathbf{M}_1)}$$

where the category 1 patterns are distributed with mean and covariance $\mathbf{M}_1$ and $\mathbf{\Sigma}_1$, respectively, and the category 2 patterns are distributed with mean and covariance $\mathbf{M}_2$ and $\mathbf{\Sigma}_2$.

The result of the comparison isn't changed if we compare logarithms instead. After some manipulation, our decision rule is then:

Decide category 1 iff:

$$(\mathbf{X}-\mathbf{M}_1)^t \mathbf{\Sigma}_1^{-1}(\mathbf{X}-\mathbf{M}_1) < (\mathbf{X}-\mathbf{M}_2)^t \mathbf{\Sigma}_2^{-1}(\mathbf{X}-\mathbf{M}_2) + B$$

where B, a constant bias term, incorporates the logarithms of the fractions preceding the exponential, *etc.*

When the quadratic forms are multiplied out and represented in terms of the components x_i, the decision rule involves a quadric surface (a hyperquadric) in n-dimensional space. The exact shape and position of this hyperquadric is determined by the means and the covariance matrices. The surface separates the space into two parts, one of which contains points that will be assigned to category 1 and the other contains points that will be assigned to category 2.

It is interesting to look at a special case of this surface. If the covariance matrices for each category are identical and diagonal, with all σ_{ii} equal to each other, then the contours of equal probability for each of the two distributions are hyperspherical. The quadric forms then become $(1/|\mathbf{\Sigma}|)(\mathbf{X}-\mathbf{M}_i)^t(\mathbf{X}-\mathbf{M}_i)$, and the decision rule is:

Decide category 1 iff:

$$(\mathbf{X}-\mathbf{M}_1)^t(\mathbf{X}-\mathbf{M}_1) < (\mathbf{X}-\mathbf{M}_2)^t(\mathbf{X}-\mathbf{M}_2)$$

Multiplying out yields:

$$\mathbf{X}\bullet\mathbf{X} - 2\mathbf{X}\bullet\mathbf{M}_1 + \mathbf{M}_1\bullet\mathbf{M}_1 < \mathbf{X}\bullet\mathbf{X} - 2\mathbf{X}\bullet\mathbf{M}_2 + \mathbf{M}_2\bullet\mathbf{M}_2$$

or finally,

Decide category 1 iff:

$$\mathbf{X}\bullet\mathbf{M}_1 \geq \mathbf{X}\bullet\mathbf{M}_2 + \text{Constant}$$

or

$$\mathbf{X}\bullet(\mathbf{M}_1 - \mathbf{M}_2) \geq \text{Constant}$$

where the constant depends on the lengths of the mean vectors.

We see that the optimal decision surface in this special case is a hyperplane. In fact, the hyperplane is perpendicular to the line joining the two means. The weights in a TLU implementation are equal to the difference in the mean vectors.

If the parameters $(\mathbf{M}_i, \mathbf{\Sigma}_i)$ of the probability distributions of the categories are not known, there are various techniques for estimating them, and then using those estimates in the decision rule. For example, if there are sufficient training patterns, one can use sample means and sample covariance matrices. (Caution: the sample covariance matrix will be singular if the training patterns happen to lie on a subspace of the whole n-dimensional space—as they certainly will, for example, if the number of training patterns is less than n.)

5.1.3 Conditionally Independent Binary Components

Suppose the vector $\mathbf{X}$ is a random variable having binary (0,1) components. We continue to denote the two probability distributions by $p(\mathbf{X} \mid 1)$ and $p(\mathbf{X} \mid 2)$. Further suppose that the components of these vectors are conditionally independent given the category. By conditional independence in this case, we mean that the formulas for the distribution can be expanded as follows:

$$p(\mathbf{X} \mid i) = p(x_1 \mid i)p(x_2 \mid i) \cdots p(x_n \mid i)$$

for $i = 1, 2$

Recall the minimum-average-loss decision rule,

Decide category 1 iff:

$$\lambda(1 \mid 2)p(\mathbf{X} \mid 2)p(2) \leq \lambda(2 \mid 1)p(\mathbf{X} \mid 1)p(1)$$

Assuming conditional independence of the components and that $\lambda(1 \mid 2) = \lambda(2 \mid 1)$, we obtain,

Decide category 1 iff:

$$p(1)p(x_1 \mid 1)p(x_2 \mid 1) \cdots p(x_n \mid 1) \geq p(x_1 \mid 2)p(x_2 \mid 2) \cdots p(x_n \mid 2)p(2)$$

or iff:

$$\frac{p(x_1 \mid 1)p(x_2 \mid 1) \ldots p(x_n \mid 1)}{p(x_1 \mid 2)p(x_2 \mid 2) \ldots p(x_n \mid 2)} \geq \frac{p(2)}{p(1)}$$

or iff:

$$\log \frac{p(x_1 \mid 1)}{p(x_1 \mid 2)} + \log \frac{p(x_2 \mid 1)}{p(x_2 \mid 2)} + \cdots + \log \frac{p(x_n \mid 1)}{p(x_n \mid 2)} + \log \frac{p(1)}{p(2)} \geq 0$$

Let us define values of the components of the distribution for specific values of their arguments, x_i :

$p(x_i = 1 \mid 1) = p_i$
$p(x_i = 0 \mid 1) = 1 - p_i$
$p(x_i = 1 \mid 2) = q_i$
$p(x_i = 0 \mid 2) = 1 - q_i$

Now, we note that since x_i can only assume the values of 1 or 0:

$$\log \frac{p(x_i \mid 1)}{p(x_i \mid 2)} = x_i \log \frac{p_i}{q_i} + (1 - x_i) \log \frac{(1 - p_i)}{(1 - q_i)}$$

$$= x_i \log \frac{p_i(1 - q_i)}{q_i(1 - p_i)} + \log \frac{(1 - p_i)}{(1 - q_i)}$$

Substituting these expressions into our decision rule yields:

Decide category 1 iff:

$$\sum_{i=1}^{n} x_i \log \frac{p_i(1-q_i)}{q_i(1-p_i)} + \sum_{i=1}^{n} \log \frac{(1-p_i)}{(1-q_i)} + \log \frac{p(1)}{p(2)} \geq 0$$

We see that we can achieve this decision with a TLU with weight values as follows:

$$w_i = \log \frac{p_i(1-q_i)}{q_i(1-p_i)}$$

for $i = 1, \ldots, n$, and

$$w_{n+1} = \log \frac{p(1)}{1-p(1)} + \sum_{i=1}^{n} \log \frac{(1-p_i)}{(1-q_i)}$$

If we do not know the p_i, q_i and $p(1)$, we can use a sample of labeled training patterns to estimate these parameters.

5.2 Learning Belief Networks

To be added.

5.3 Nearest-Neighbor Methods

Another class of methods can be related to the statistical ones. These are called *nearest-neighbor* methods or, sometimes, *memory-based* methods. (A collection of papers on this subject is in [Dasarathy, 1991].) Given a training set Ξ of m labeled patterns, a nearest-neighbor procedure decides that some new pattern, $\mathbf{X}$, belongs to the same category as do its closest neighbors in Ξ. More precisely, a k-nearest-neighbor method assigns a new pattern, $\mathbf{X}$, to that category to which the plurality of its k closest neighbors belong. Using relatively large values of k decreases the chance that the decision will be unduly influenced by a noisy training pattern close to $\mathbf{X}$. But large values of k also reduce the acuity of the method. The k-nearest-neighbor method can be thought of as estimating the values of the probabilities of the classes given $\mathbf{X}$. Of course the denser are the points around $\mathbf{X}$, and the larger the value of k, the better the estimate.

The distance metric used in nearest-neighbor methods (for numerical attributes) can be simple Euclidean distance. That is, the distance between two patterns $(x_{11}, x_{12}, \ldots, x_{1n})$ and $(x_{21}, x_{22}, \ldots, x_{2n})$ is $\sqrt{\sum_{j=1}^{n}(x_{1j} - x_{2j})^2}$. This distance measure is often modified by *scaling* the features so that the spread of attribute values along each dimension is approximately the same. In that case, the distance between the two vectors would be $\sqrt{\sum_{j=1}^{n} a_j^2(x_{1j} - x_{2j})^2}$, where a_j is the scale factor for dimension j.

An example of a nearest-neighbor decision problem is shown in Fig. 5.3. In the figure the class of a training pattern is indicated by the number next to it.

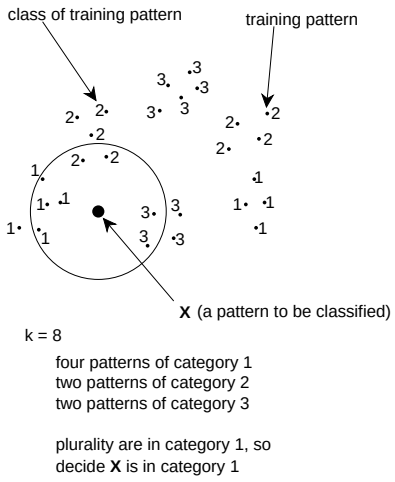

class of training pattern
training pattern
X (a pattern to be classified)
k = 8
four patterns of category 1
two patterns of category 2
two patterns of category 3
plurality are in category 1, so
decide X is in category 1

Figure 5.3: An 8-Nearest-Neighbor Decision

See [Baum, 1994] for theoretical analysis of error rate as a function of the number of training patterns for the case in which points are randomly distributed on the surface of a unit sphere and underlying function is linearly separable.

Nearest-neighbor methods are memory intensive because a large number of training patterns must be stored to achieve good generalization. Since memory cost is now reasonably low, the method and its derivatives have seen several practical applications. (See, for example, [Moore, 1992, Moore, *et al.*, 1994]. Also, the distance calculations required to find nearest neighbors can often be efficiently computed by *kd-tree* methods [Friedman, *et al.*, 1977].

A theorem by Cover and Hart [Cover & Hart, 1967] relates the performance of the 1-nearest-neighbor method to the performance of a minimum-

probability-of-error classifier. As mentioned earlier, the minimum-probability-of-error classifier would assign a new pattern $\mathbf{X}$ to that category that maximized $p(i)p(\mathbf{X} \mid i)$, where $p(i)$ is the a priori probability of category i, and $p(\mathbf{X} \mid i)$ is the probability (or probability density function) of $\mathbf{X}$ given that $\mathbf{X}$ belongs to category i, for categories $i = 1, \ldots, R$. Suppose the probability of error in classifying patterns of such a minimum-probability-of-error classifier is ε. The Cover-Hart theorem states that under very mild conditions (having to do with the smoothness of probability density functions) the probability of error, ε_{nn}, of a 1-nearest-neighbor classifier is bounded by:

$$\varepsilon \leq \varepsilon_{nn} \leq \varepsilon\left(2 - \varepsilon\frac{R}{R-1}\right) \leq 2\varepsilon$$

where R is the number of categories.

Also see [Aha, 1991].

5.4 Bibliographical and Historical Remarks

To be added.

Chapter 6

Decision Trees

6.1 Definitions

A *decision tree* (generally defined) is a tree whose internal nodes are tests (on input patterns) and whose leaf nodes are categories (of patterns). We show an example in Fig. 6.1. A decision tree assigns a class number (or output) to an input pattern by filtering the pattern down through the tests in the tree. Each test has mutually exclusive and exhaustive outcomes. For example, test T_2 in the tree of Fig. 6.1 has three outcomes; the left-most one assigns the input pattern to class 3, the middle one sends the input pattern down to test T_4, and the right-most one assigns the pattern to class 1. We follow the usual convention of depicting the leaf nodes by the class number.[1] Note that in discussing decision trees we are not limited to implementing Boolean functions—they are useful for general, categorically valued functions.

There are several dimensions along which decision trees might differ:

a. The tests might be *multivariate* (testing on several features of the input at once) or *univariate* (testing on only one of the features).

b. The tests might have two outcomes or more than two. (If all of the tests have two outcomes, we have a *binary decision tree*.)

[1] One of the researchers who has done a lot of work on learning decision trees is Ross Quinlan. Quinlan distinguishes between classes and categories. He calls the subsets of patterns that filter down to each tip *categories* and subsets of patterns having the same label *classes*. In Quinlan's terminology, our example tree has nine categories and three classes. We will not make this distinction, however, but will use the words "category" and "class" interchangeably to refer to what Quinlan calls "class."

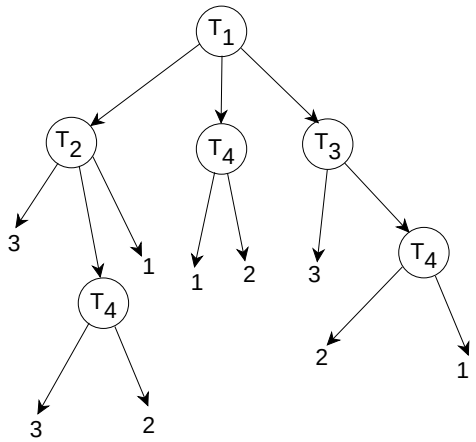


Figure 6.1: A Decision Tree

c. The features or attributes might be categorical or numeric. (Binary-valued ones can be regarded as either.)

d. We might have two classes or more than two. If we have two classes and binary inputs, the tree implements a Boolean function, and is called a Boolean decision tree.

It is straightforward to represent the function implemented by a univariate Boolean decision tree in DNF form. The DNF form implemented by such a tree can be obtained by tracing down each path leading to a tip node corresponding to an output value of 1, forming the conjunction of the tests along this path, and then taking the disjunction of these conjunctions. We show an example in Fig. 6.2. In drawing univariate decision trees, each non-leaf node is depicted by a single attribute. If the attribute has value 0 in the input pattern, we branch left; if it has value 1, we branch right.

The k-DL class of Boolean functions can be implemented by a multivariate decision tree having the (highly unbalanced) form shown in Fig. 6.3. Each test, c_i, is a term of size k or less. The v_i all have values of 0 or 1.

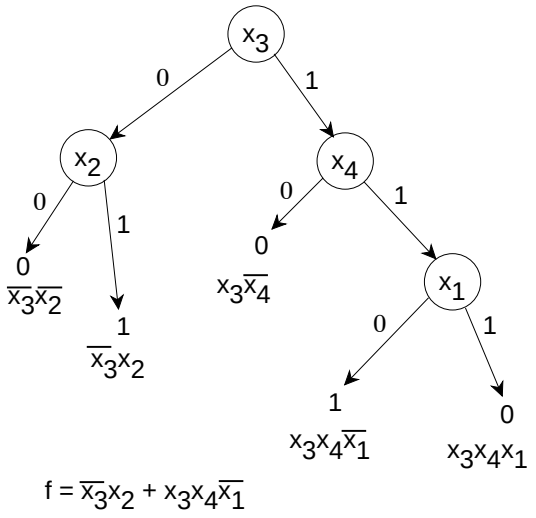


Figure 6.2: A Decision Tree Implementing a DNF Function

6.2 Supervised Learning of Univariate Decision Trees

Several systems for learning decision trees have been proposed. Prominent among these are ID3 and its new version, C4.5 [Quinlan, 1986, Quinlan, 1993], and CART [Breiman, *et al.*, 1984] We discuss here only batch methods, although incremental ones have also been proposed [Utgoff, 1989].

6.2.1 Selecting the Type of Test

As usual, we have n features or attributes. If the attributes are binary, the tests are simply whether the attribute's value is 0 or 1. If the attributes are categorical, but non-binary, the tests might be formed by dividing the attribute values into mutually exclusive and exhaustive subsets. A decision tree with such tests is shown in Fig. 6.4. If the attributes are numeric, the tests might involve "interval tests," for example $7 \leq x_i \leq 13.2$.

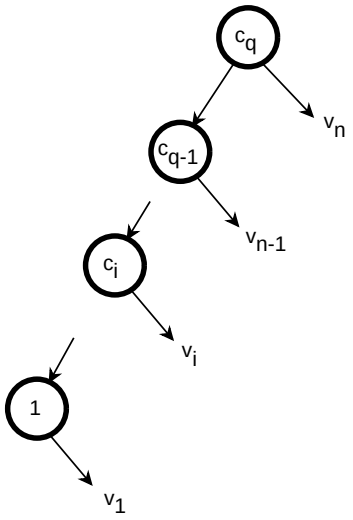


Figure 6.3: A Decision Tree Implementing a Decision List

6.2.2 Using Uncertainty Reduction to Select Tests

The main problem in learning decision trees for the binary-attribute case is selecting the order of the tests. For categorical and numeric attributes, we must also decide what the tests should be (besides selecting the order). Several techniques have been tried; the most popular one is at each stage to select that test that maximally reduces an entropy-like measure.

We show how this technique works for the simple case of tests with binary outcomes. Extension to multiple-outcome tests is straightforward computationally but gives poor results because entropy is always decreased by having more outcomes.

The *entropy* or uncertainty still remaining about the class of a pattern—knowing that it is in some set, Ξ, of patterns is defined as:

$$H(\Xi) = -\sum_i p(i|\Xi) \log_2 p(i|\Xi)$$

where $p(i|\Xi)$ is the probability that a pattern drawn at random from Ξ belongs to class i, and the summation is over all of the classes. We want to select tests at each node such that as we travel down the decision tree, the uncertainty about the class of a pattern becomes less and less.

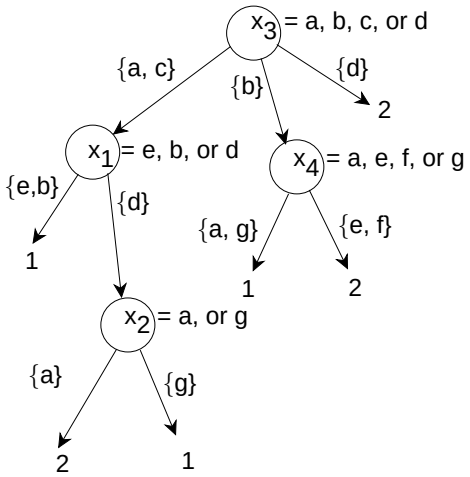


Figure 6.4: A Decision Tree with Categorical Attributes

Since we do not in general have the probabilities $p(i|\Xi)$, we estimate them by sample statistics. Although these estimates might be errorful, they are nevertheless useful in estimating uncertainties. Let $\hat{p}(i|\Xi)$ be the number of patterns in Ξ belonging to class i divided by the total number of patterns in Ξ. Then an estimate of the uncertainty is:

$$\hat{H}(\Xi) = -\sum_i \hat{p}(i|\Xi) \log_2 \hat{p}(i|\Xi)$$

For simplicity, from now on we'll drop the "hats" and use sample statistics as if they were real probabilities.

If we perform a test, T, having k possible outcomes on the patterns in Ξ, we will create k subsets, $\Xi_1, \Xi_2, \ldots, \Xi_k$. Suppose that n_i of the patterns in Ξ are in Ξ_i for $i = 1, ..., k$. (Some n_i may be 0.) If we knew that T applied to a pattern in Ξ resulted in the j-th outcome (that is, we knew that the pattern was in Ξ_j), the uncertainty about its class would be:

$$H(\Xi_j) = -\sum_i p(i|\Xi_j) \log_2 p(i|\Xi_j)$$

and the *reduction* in uncertainty (beyond knowing only that the pattern was in Ξ) would be:

$$H(\Xi) - H(\Xi_j)$$

Of course we cannot say that the test T is guaranteed always to produce that amount of reduction in uncertainty because we don't know that the result of the test will be the j-th outcome. But we can estimate the *average* uncertainty over all the Ξ_j, by:

$$E[H_T(\Xi)] = \sum_j p(\Xi_j) H(\Xi_j)$$

where by $H_T(\Xi)$ we mean the average uncertainty after performing test T on the patterns in Ξ, $p(\Xi_j)$ is the probability that the test has outcome j, and the sum is taken from 1 to k. Again, we don't know the probabilities $p(\Xi_j)$, but we can use sample values. The estimate $\hat{p}(\Xi_j)$ of $p(\Xi_j)$ is just the number of those patterns in Ξ that have outcome j divided by the total number of patterns in Ξ. The *average* reduction in uncertainty achieved by test T (applied to patterns in Ξ) is then:

$$R_T(\Xi) = H(\Xi) - E[H_T(\Xi)]$$

An important family of decision tree learning algorithms selects for the root of the tree that test that gives maximum reduction of uncertainty, and then applies this criterion recursively until some termination condition is met (which we shall discuss in more detail later). The uncertainty calculations are particularly simple when the tests have binary outcomes and when the attributes have binary values. We'll give a simple example to illustrate how the test selection mechanism works in that case.

Suppose we want to use the uncertainty-reduction method to build a decision tree to classify the following patterns:

pattern	class
(0, 0, 0)	0
(0, 0, 1)	0
(0, 1, 0)	0
(0, 1, 1)	0
(1, 0, 0)	0
(1, 0, 1)	1
(1, 1, 0)	0
(1, 1, 1)	1

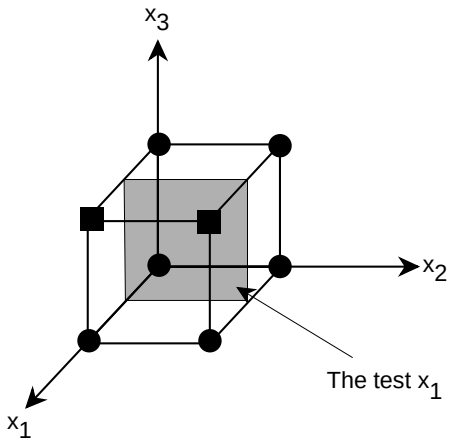


Figure 6.5: Eight Patterns to be Classified by a Decision Tree

What single test, x_1, x_2, or x_3, should be performed first? The illustration in Fig. 6.5 gives geometric intuition about the problem.

The initial uncertainty for the set, Ξ, containing all eight points is:

$$H(\Xi) = -(6/8)\log_2(6/8) - (2/8)\log_2(2/8) = 0.81$$

Next, we calculate the uncertainty reduction if we perform x_1 first. The left-hand branch has only patterns belonging to class 0 (we call them the set Ξ_l), and the right-hand-branch (Ξ_r) has two patterns in each class. So, the uncertainty of the left-hand branch is:

$$H_{x_1}(\Xi_l) = -(4/4)\log_2(4/4) - (0/4)\log_2(0/4) = 0$$

And the uncertainty of the right-hand branch is:

$$H_{x_1}(\Xi_r) = -(2/4)\log_2(2/4) - (2/4)\log_2(2/4) = 1$$

Half of the patterns "go left" and half "go right" on test x_1. Thus, the average uncertainty after performing the x_1 test is:

$$1/2 H_{x_1}(\Xi_l) + 1/2 H_{x_1}(\Xi_r) = 0.5$$

Therefore the uncertainty reduction on Ξ achieved by x_1 is:

$$R_{x_1}(\Xi) = 0.81 - 0.5 = 0.31$$

By similar calculations, we see that the test x_3 achieves exactly the same uncertainty reduction, but x_2 achieves no reduction whatsoever. Thus, our "greedy" algorithm for selecting a first test would select either x_1 or x_3. Suppose x_1 is selected. The uncertainty-reduction procedure would select x_3 as the next test. The decision tree that this procedure creates thus implements the Boolean function: $f = x_1 x_3$.

See [Quinlan, 1986, sect. 4] for another example.

6.2.3 Non-Binary Attributes

If the attributes are non-binary, we can still use the uncertainty-reduction technique to select tests. But now, in addition to selecting an attribute, we must select a test on that attribute. Suppose for example that the value of an attribute is a real number and that the test to be performed is to set a threshold and to test to see if the number is greater than or less than that threshold. In principle, given a set of labeled patterns, we can measure the uncertainty reduction for each test that is achieved by every possible threshold (there are only a finite number of thresholds that give different test results if there are only a finite number of training patterns). Similarly, if an attribute is categorical (with a finite number of categories), there are only a finite number of mutually exclusive and exhaustive subsets into which the values of the attribute can be split. We can calculate the uncertainty reduction for each split.

6.3 Networks Equivalent to Decision Trees

Since univariate Boolean decision trees are implementations of DNF functions, they are also equivalent to two-layer, feedforward neural networks. We show an example in Fig. 6.6. The decision tree at the left of the figure implements the same function as the network at the right of the figure. Of course, when implemented as a network, all of the features are evaluated in parallel for any input pattern, whereas when implemented as a decision tree only those features on the branch traveled down by the input pattern need to be evaluated. The decision-tree induction methods discussed in this chapter can thus be thought of as particular ways to establish the structure and the weight values for networks.

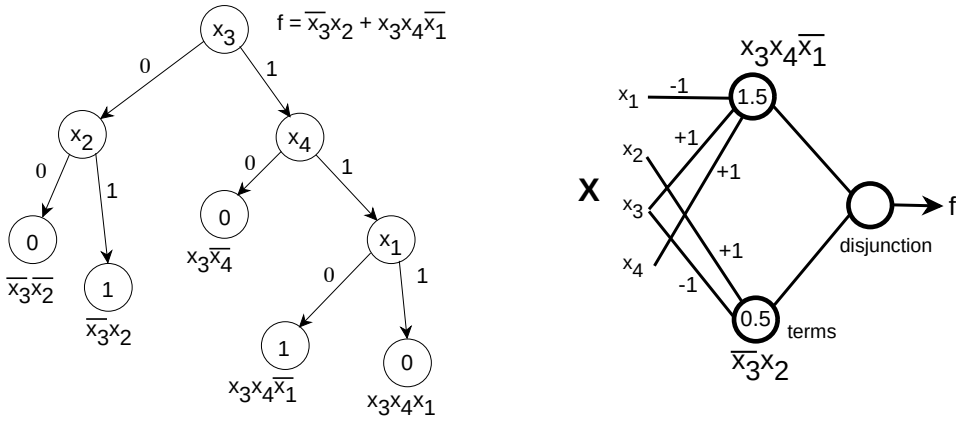


Figure 6.6: A Univariate Decision Tree and its Equivalent Network

Multivariate decision trees with linearly separable functions at each node can also be implemented by feedforward networks—in this case three-layer ones. We show an example in Fig. 6.7 in which the linearly separable functions, each implemented by a TLU, are indicated by L_1, L_2, L_3, and L_4. Again, the final layer has fixed weights, but the weights in the first two layers must be trained. Different approaches to training procedures have been discussed by [Brent, 1990], by [John, 1995], and (for a special case) by [Marchand & Golea, 1993].

6.4 Overfitting and Evaluation

6.4.1 Overfitting

In supervised learning, we must choose a function to fit the training set from among a set of hypotheses. We have already showed that generalization is impossible without bias. When we know a priori that the function we are trying to guess belongs to a small subset of all possible functions, then, even with an incomplete set of training samples, it is possible to reduce the subset of functions that are consistent with the training set sufficiently to make useful guesses about the value of the function for inputs not in the training set. And, the larger the training set, the more likely it is that even a randomly selected consistent function will have appropriate outputs for

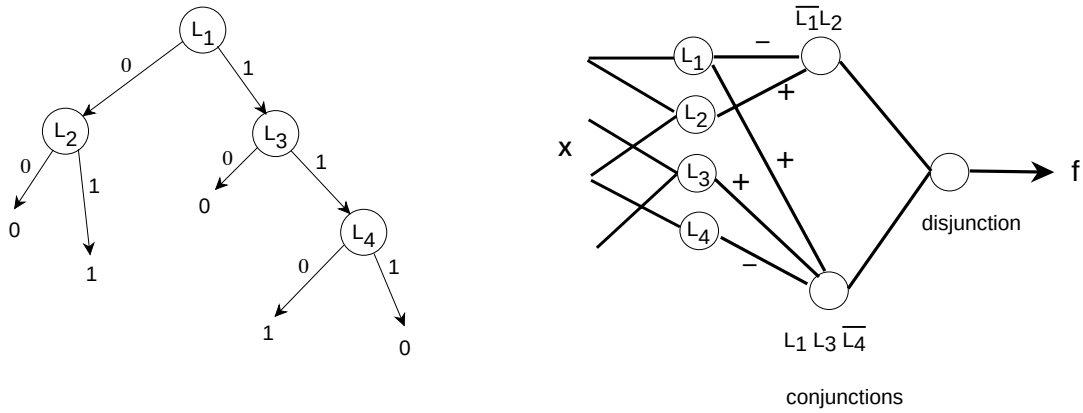


Figure 6.7: A Multivariate Decision Tree and its Equivalent Network

patterns not yet seen.

However, even with bias, if the training set is not sufficiently large compared with the size of the hypothesis space, there will still be too many consistent functions for us to make useful guesses, and generalization performance will be poor. When there are too many hypotheses that are consistent with the training set, we say that we are *overfitting* the training data. Overfitting is a problem that we must address for all learning methods.

Since a decision tree of sufficient size can implement *any* Boolean function there is a danger of overfitting—especially if the training set is small. That is, even if the decision tree is synthesized to classify all the members of the training set correctly, it might perform poorly on new patterns that were not used to build the decision tree. Several techniques have been proposed to avoid overfitting, and we shall examine some of them here. They make use of methods for estimating how well a given decision tree might generalize—methods we shall describe next.

6.4.2 Validation Methods

The most straightforward way to estimate how well a hypothesized function (such as a decision tree) performs on a test set is to test it on the test set! But, if we are comparing several learning systems (for example, if we are comparing different decision trees) so that we can select the one

that performs the best on the test set, then such a comparison amounts to "training on the test data." True, training on the test data enlarges the training set, with a consequent expected improvement in generalization, but there is still the danger of overfitting if we are comparing several different learning systems. Another technique is to split the training set—using (say) two-thirds for training and the other third for estimating generalization performance. But splitting reduces the size of the training set and thereby increases the possibility of overfitting. We next describe some validation techniques that attempt to avoid these problems.

Cross-Validation

In *cross-validation*, we divide the training set Ξ into K mutually exclusive and exhaustive equal-sized subsets: $\Xi_1, \ldots, \Xi_K$. For each subset, Ξ_i, train on the union of all of the other subsets, and empirically determine the error rate, ε_i, on Ξ_i. (The error rate is the number of classification errors made on Ξ_i divided by the number of patterns in Ξ_i.) An estimate of the error rate that can be expected on new patterns of a classifier trained on *all* the patterns in Ξ is then the average of the ε_i.

Leave-one-out Validation

Leave-one-out validation is the same as cross validation for the special case in which K equals the number of patterns in Ξ, and each Ξ_i consists of a single pattern. When testing on each Ξ_i, we simply note whether or not a mistake was made. We count the total number of mistakes and divide by K to get the estimated error rate. This type of validation is, of course, more expensive computationally, but useful when a more accurate estimate of the error rate for a classifier is needed.

Describe "bootstrapping" also [Efron, 1982].

6.4.3 Avoiding Overfitting in Decision Trees

Near the tips of a decision tree there may be only a few patterns per node. For these nodes, we are selecting a test based on a very small sample, and thus we are likely to be overfitting. This problem can be dealt with by terminating the test-generating procedure before all patterns are perfectly split into their separate categories. That is, a leaf node may contain patterns of more than one class, but we can decide in favor of the most numerous class. This procedure will result in a few errors but often accepting a small number of errors on the training set results in fewer errors on a testing set.

This behavior is illustrated in Fig. 6.8.

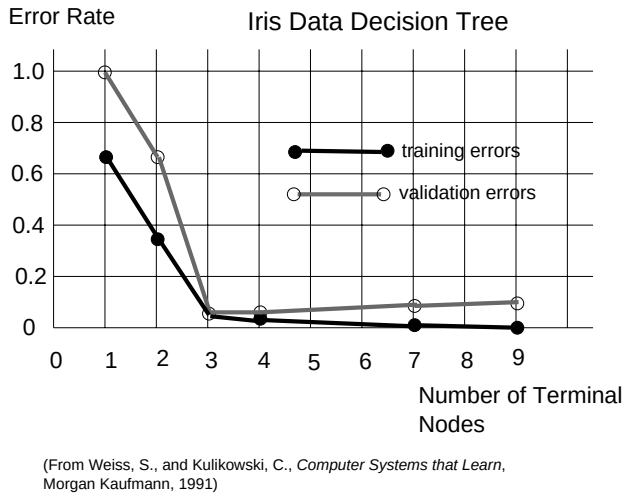


Figure 6.8: Determining When Overfitting Begins

One can use cross-validation techniques to determine when to stop splitting nodes. If the cross validation error increases as a consequence of a node split, then don't split. One has to be careful about when to stop, though, because underfitting usually leads to more errors on test sets than does overfitting. There is a general rule that the lowest error-rate attainable by a sub-tree of a fully expanded tree can be no less than 1/2 of the error rate of the fully expanded tree [Weiss & Kulikowski, 1991, page 126].

Rather than stopping the growth of a decision tree, one might grow it to its full size and then prune away leaf nodes and their ancestors until cross-validation accuracy no longer increases. This technique is called *post-pruning*. Various techniques for pruning are discussed in [Weiss & Kulikowski, 1991].

6.4.4 Minimum-Description Length Methods

An important tree-growing and pruning technique is based on the *minimum-description-length (MDL)* principle. (MDL is an important idea that extends beyond decision-tree methods [Rissanen, 1978].) The idea is that the simplest decision tree that can predict the classes of the training patterns is the best one. Consider the problem of transmitting just the

labels of a training set of patterns, assuming that the receiver of this information already has the ordered set of patterns. If there are m patterns, each labeled by one of R classes, one could transmit a list of m R-valued numbers. Assuming equally probable classes, this transmission would require $m \log_2 R$ bits. Or, one could transmit a decision tree that correctly labelled all of the patterns. The number of bits that this transmission would require depends on the technique for encoding decision trees and on the size of the tree. If the tree is small and accurately classifies all of the patterns, it might be more economical to transmit the tree than to transmit the labels directly. In between these extremes, we might transmit a tree plus a list of labels of all the patterns that the tree misclassifies.

In general, the number of bits (or description length of the binary encoded message) is $t + d$, where t is the length of the message required to transmit the tree, and d is the length of the message required to transmit the labels of the patterns misclassified by the tree. In a sense, that tree associated with the smallest value of $t+d$ is the best or most economical tree. The MDL method is one way of adhering to the Occam's razor principle.

Quinlan and Rivest [Quinlan & Rivest, 1989] have proposed techniques for encoding decision trees and lists of exception labels and for calculating the description length $(t + d)$ of these trees and labels. They then use the description length as a measure of quality of a tree in two ways:

a. In growing a tree, they use the reduction in description length to select tests (instead of reduction in uncertainty).

b. In pruning a tree after it has been grown to zero error, they prune away those nodes (starting at the tips) that achieve a decrease in the description length.

These techniques compare favorably with the uncertainty-reduction method, although they are quite sensitive to the coding schemes used.

6.4.5 Noise in Data

Noise in the data means that one must inevitably accept some number of errors—depending on the noise level. Refusal to tolerate errors on the training set when there is noise leads to the problem of "fitting the noise." Dealing with noise, then, requires accepting some errors at the leaf nodes just as does the fact that there are a small number of patterns at leaf nodes.

6.5 The Problem of Replicated Subtrees

Decision trees are not the most economical means of implementing some Boolean functions. Consider, for example, the function $f = x_1x_2 + x_3x_4$. A decision tree for this function is shown in Fig. 6.9. Notice the replicated subtrees shown circled. The DNF-form equivalent to the function implemented by this decision tree is $f = x_1x_2 + x_1\overline{x_2}x_3x_4 + \overline{x_1}x_3x_4$. This DNF form is non-minimal (in the number of disjunctions) and is equivalent to $f = x_1x_2 + x_3x_4$.

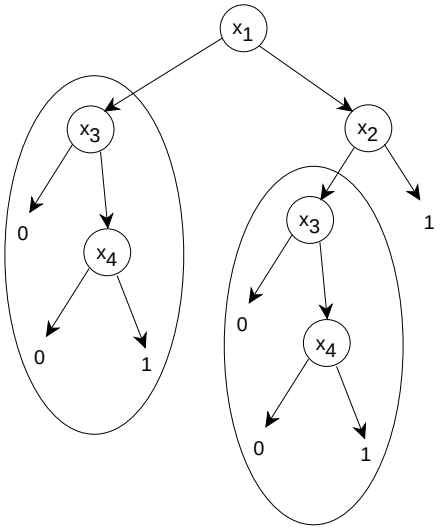


Figure 6.9: A Decision Tree with Subtree Replication

The need for replication means that it takes longer to learn the tree and that subtrees replicated further down the tree must be learned using a smaller training subset. This problem is sometimes called the *fragmentation problem.*

Several approaches might be suggested for dealing with fragmentation. One is to attempt to build a *decision graph* instead of a tree [Oliver, Dowe, & Wallace, 1992, Kohavi, 1994]. A decision graph that implements the same decisions as that of the decision tree of Fig. 6.9 is shown in Fig. 6.10.

Another approach is to use multivariate (rather than univariate tests at each node). In our example of learning $f = x_1x_2 + x_3x_4$, if we had a test

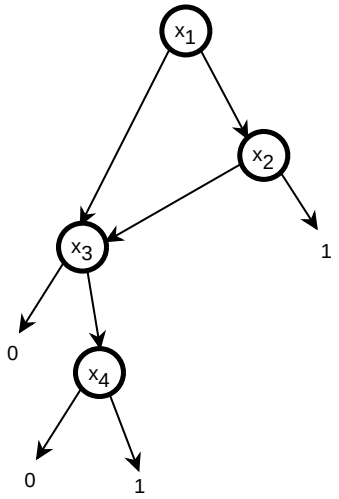


Figure 6.10: A Decision Graph

for x_1x_2 and a test for x_3x_4, the decision tree could be much simplified, as shown in Fig. 6.11. Several researchers have proposed techniques for learning decision trees in which the tests at each node are linearly separable functions. [John, 1995] gives a nice overview (with several citations) of learning such *linear discriminant trees* and presents a method based on "soft entropy."

A third method for dealing with the replicated subtree problem involves extracting propositional "rules" from the decision tree. The rules will have as antecedents the conjunctions that lead down to the leaf nodes, and as consequents the name of the class at the corresponding leaf node. An example rule from the tree with the repeating subtree of our example would be: $x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4 \supset 1$. Quinlan [Quinlan, 1987] discusses methods for reducing a set of rules to a simpler set by 1) eliminating from the antecedent of each rule any "unnecessary" conjuncts, and then 2) eliminating "unnecessary" rules. A conjunct or rule is determined to be unnecessary if its elimination has little effect on classification accuracy—as determined by a chi-square test, for example. After a rule set is processed, it might be the case that more than one rule is "active" for any given pattern, and care must be taken that the active rules do not conflict in their decision about the class of a pattern.

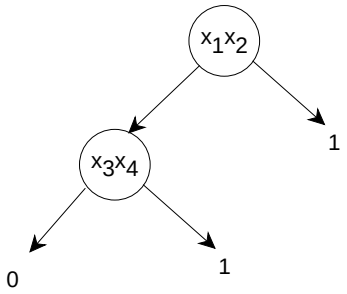


Figure 6.11: A Multivariate Decision Tree

6.6 The Problem of Missing Attributes

To be added.

6.7 Comparisons

Several experimenters have compared decision-tree, neural-net, and nearest-neighbor classifiers on a wide variety of problems. For a comparison of neural nets versus decision trees, for example, see [Dietterich, *et al.*, 1990, Shavlik, Mooney, & Towell, 1991, Quinlan, 1994]. In their *StatLog* project, [Taylor, Michie, & Spiegalhalter, 1994] give thorough comparisons of several machine learning algorithms on several different types of problems. There seems to be no single type of classifier that is best for all problems. And, there do not seem to be any general conclusions that would enable one to say which classifier method is best for which sorts of classification problems, although [Quinlan, 1994] does provide some intuition about properties of problems that might render them ill suited for decision trees, on the one hand, or backpropagation, on the other.

6.8 Bibliographical and Historical Remarks

To be added.

Chapter 7

Inductive Logic Programming

There are many different representational forms for functions of input variables. So far, we have seen (Boolean) algebraic expressions, decision trees, and neural networks, plus other computational mechanisms such as techniques for computing nearest neighbors. Of course, the representation most important in computer science is a computer program. For example, a Lisp predicate of binary-valued inputs computes a Boolean function of those inputs. Similarly, a logic program (whose ordinary application is to compute bindings for variables) can also be used simply to decide whether or not a predicate has value *True (T)* or *False (F)*. For example, the Boolean exclusive-or (odd parity) function of two variables can be computed by the following logic program:

```
Parity(x,y) :- True(x), ¬ True(y)
            :- True(y), ¬ True(x)
```

We follow Prolog syntax (see, for example, [Mueller & Page, 1988]), except that our convention is to write variables as strings beginning with lower-case letters and predicates as strings beginning with upper-case letters. The unary function "`True`" returns T if and only if the value of its argument is T. (We now think of Boolean functions and arguments as having values of T and F instead of 0 and 1.) Programs will be written in "`typewriter`" font.

In this chapter, we consider the matter of learning logic programs given a set of variable values for which the logic program should return T (the *positive instances*) and a set of variable values for which it should return F (the *negative instances*). The subspecialty of machine learning that deals with learning logic programs is called *inductive logic programming (ILP)* [Lavrač & Džeroski, 1994]. As with any learning problem, this one can be quite complex and intractably difficult unless we constrain it with biases of some sort. In ILP, there are a variety of possible biases (called *language biases*). One might restrict the program to Horn clauses, not allow recursion, not allow functions, and so on.

As an example of an ILP problem, suppose we are trying to induce a function `Nonstop(x,y)`, that is to have value T for pairs of cities connected by a non-stop air flight and F for all other pairs of cities. We are given a training set consisting of positive and negative examples. As positive examples, we might have `(A,B)`, `(A, A1)`, and some other pairs; as negative examples, we might have `(A1, A2)`, and some other pairs. In ILP, we usually have additional information about the examples, called "background knowledge." In our air-flight problem, the background information might be such ground facts as `Hub(A)`, `Hub(B)`, `Satellite(A1,A)`, plus others. (`Hub(A)` is intended to mean that the city denoted by `A` is a hub city, and `Satellite(A1,A)` is intended to mean that the city denoted by `A1` is a satellite of the city denoted by `A`.) From these training facts, we want to induce a program `Nonstop(x,y)`, written in terms of the background relations `Hub` and `Satellite`, that has value T for all the positive instances and has value F for all the negative instances. Depending on the exact set of examples, we might induce the program:

```
Nonstop(x,y) :- Hub(x), Hub(y)
             :- Satellite(x,y)
             :- Satellite(y,x)
```

which would have value T if both of the two cities were hub cities or if one were a satellite of the other. As with other learning problems, we want the induced program to generalize well; that is, if presented with arguments not represented in the training set (but for which we have the needed background knowledge), we would like the function to guess well.

7.1 Notation and Definitions

In evaluating logic programs in ILP, we implicitly append the background facts to the program and adopt the usual convention that a program has value T for a set of inputs if and only if the program interpreter returns T when actually running the program (with background facts appended) on those inputs; otherwise it has value F. Using the given background facts, the program above would return T for input `(A, A1)`, for example. If a logic program, π, returns T for a set of arguments $\mathbf{X}$, we say that the program *covers* the arguments and write covers$(\pi, \mathbf{X})$. Following our terminology introduced in connection with version spaces, we will say that a program is *sufficient* if it covers all of the positive instances and that it is *necessary* if it does not cover any of the negative instances. (That is, a program implements a sufficient condition that a training instance is positive if it covers *all* of the positive training instances; it implements a necessary condition if it covers *none* of the negative instances.) In the noiseless case, we want to induce a program that is both sufficient and necessary, in which case we will call it *consistent*. With imperfect (noisy) training sets, we might relax this criterion and settle for a program that covers all but some fraction of the positive instances while allowing it to cover some fraction of the negative instances. We illustrate these definitions schematically in Fig. 7.1.

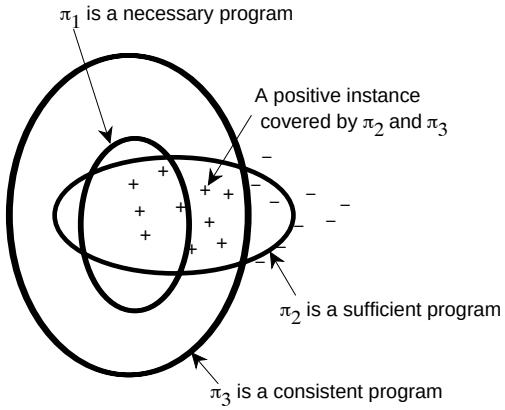

π_1 is a necessary program
A positive instance
covered by π_2 and π_3
π_2 is a sufficient program
π_3 is a consistent program

Figure 7.1: Sufficient, Necessary, and Consistent Programs

As in version spaces, if a program is sufficient but not necessary it can be

made to cover fewer examples by *specializing* it. Conversely, if it is necessary but not sufficient, it can be made to cover more examples by *generalizing* it. Suppose we are attempting to induce a logic program to compute the relation ρ. The most *general* logic program, which is certainly sufficient, is the one that has value T for *all* inputs, namely a single clause with an empty body, [ρ :-], which is called a *fact* in Prolog. The most *special* logic program, which is certainly necessary, is the one that has value F for *all* inputs, namely [ρ :- F]. Two of the many different ways to search for a consistent logic program are: 1) start with [ρ :-] and specialize until the program is consistent, or 2) start with [ρ :- F] and generalize until the program is consistent. We will be discussing a method that starts with [ρ :-], specializes until the program is necessary (but might no longer be sufficient), then reachieves sufficiency in stages by generalizing—ensuring within each stage that the program remains necessary (by specializing).

7.2 A Generic ILP Algorithm

Since the primary operators in our search for a consistent program are specialization and generalization, we must next discuss those operations. There are three major ways in which a logic program might be generalized:

a. Replace some terms in a program clause by variables. (Readers familiar with substitutions in the predicate calculus will note that this process is the inverse of substitution.)

b. Remove literals from the body of a clause.

c. Add a clause to the program

Analogously, there are three ways in which a logic program might be specialized:

a. Replace some variables in a program clause by terms (a *substitution*).

b. Add literals to the body of a clause.

c. Remove a clause from the program

We will be presenting an ILP learning method that adds clauses to a program when generalizing and that adds literals to the body of a clause when specializing. When we add a clause, we will always add the clause [ρ :-]

and then specialize it by adding literals to the body. Thus, we need only describe the process for adding literals.

Clauses can be partially ordered by the specialization relation. In general, clause c_1 is more special than clause c_2 if $c_2 \models c_1$. A special case, which is what we use here, is that a clause c_1 is more special than a clause c_2 if the set of literals in the body of c_2 is a subset of those in c_1. This ordering relation can be used in a structure of partially ordered clauses, called the *refinement graph*, that is similar to a version space. Clause c_1 is an immediate successor of clause c_2 in this graph if and only if clause c_1 can be obtained from clause c_2 by adding a literal to the body of c_2. A refinement graph then tells us the ways in which we can specialize a clause by adding a literal to it.

Of course there are unlimited possible literals we might add to the body of a clause. Practical ILP systems restrict the literals in various ways. Typical allowed additions are:

a. Literals used in the background knowledge.

b. Literals whose arguments are a subset of those in the head of the clause.

c. Literals that introduce a new distinct variable different from those in the head of the clause.

d. A literal that equates a variable in the head of the clause with another such variable or with a term mentioned in the background knowledge. (This possibility is equivalent to forming a specialization by making a substitution.)

e. A literal that is the same (except for its arguments) as that in the head of the clause. (This possibility admits recursive programs, which are disallowed in some systems.)

We can illustrate these possibilities using our air-flight example. We start with the program [`Nonstop(x,y) :-`]. The literals used in the background knowledge are `Hub` and `Satellite`. Thus the literals that we might consider adding are:

```
Hub(x)
Hub(y)
Hub(z)
Satellite(x,y)
```

```
Satellite(y,x)
Satellite(x,z)
Satellite(z,y)
(x = y)
```

(If recursive programs are allowed, we could also add the literals `Nonstop(x,z)` and `Nonstop(z,y)`.) These possibilities are among those illustrated in the refinement graph shown in Fig. 7.2. Whatever restrictions on additional literals are imposed, they are all syntactic ones from which the successors in the refinement graph are easily computed. ILP programs that follow the approach we are discussing (of specializing clauses by adding a literal) thus have well defined methods of computing the possible literals to add to a clause.

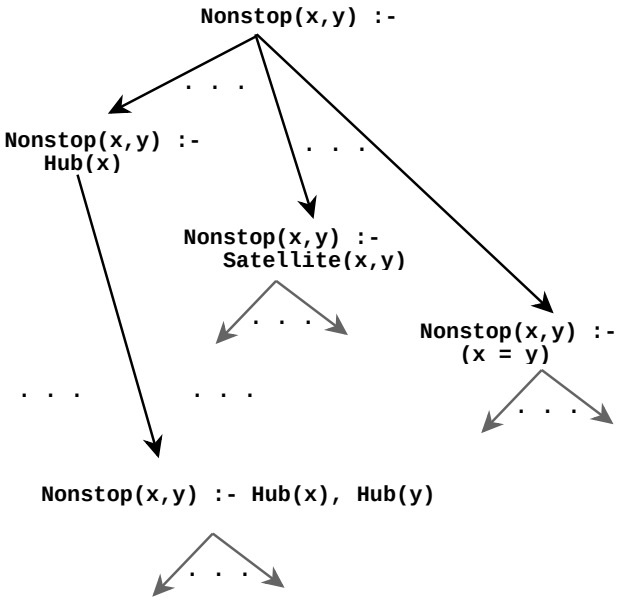


Figure 7.2: Part of a Refinement Graph

Now we are ready to write down a simple generic algorithm for inducing a logic program, π for inducing a relation ρ. We are given a training set, Ξ of argument sets some known to be in the relation ρ and some not in

ρ; Ξ^+ are the positive instances, and Ξ^- are the negative instances. The algorithm has an outer loop in which it successively adds clauses to make π more and more sufficient. It has an inner loop for constructing a clause, c, that is more and more necessary and in which it refers only to a subset, Ξ_{cur}, of the training instances. (The positive instances in Ξ_{cur} will be denoted by Ξ^+_{cur}, and the negative ones by Ξ^-_{cur}.) The algorithm is also given background relations and the means for adding literals to a clause. It uses a logic program interpreter to compute whether or not the program it is inducing covers training instances. The algorithm can be written as follows:

Generic ILP Algorithm

(Adapted from [Lavrač & Džeroski, 1994, p. 60].)

Initialize $\Xi_{cur} := \Xi$.
Initialize $\pi :=$ empty set of clauses.
repeat [The outer loop works to make π sufficient.]
    Initialize $c := \rho :- .$
    repeat [The inner loop makes c necessary.]
        Select a literal l to add to c. [This is a nondeterministic choice point.]
        Assign $c := c, l$.
    until c is necessary. [That is, until c covers no negative instances in Ξ_{cur}.]
    Assign $\pi := \pi, c$. [We add the clause c to the program.]
    Assign $\Xi_{cur} := \Xi_{cur} -$ (the positive instances in Ξ_{cur} covered by π).
until π is sufficient.

(The termination tests for the inner and outer loops can be relaxed as appropriate for the case of noisy instances.)

7.3 An Example

We illustrate how the algorithm works by returning to our example of airline flights. Consider the portion of an airline route map, shown in Fig. 7.3. Cities A, B, and C are "hub" cities, and we know that there are nonstop flights between all hub cities (even those not shown on this portion of the route map). The other cities are "satellites" of one of the hubs, and we know that there are nonstop flights between each satellite city and its hub. The learning program is given a set of positive instances, Ξ^+, of pairs of cities between which there are nonstop flights and a set of negative instances, Ξ^-,

of pairs of cities between which there are not nonstop flights. Ξ^+ contains just the pairs:

$$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >,$$
$$< A, A1 >, < A, A2 >, < A1, A >, < A2, A >, < B, B1 >, < B, B2 >,$$
$$< B1, B >, < B2, B >, < C, C1 >, < C, C2 >, < C1, C >, < C2, C >\}$$

For our example, we will assume that Ξ^- contains all those pairs of cities shown in Fig. 7.3 that are not in Ξ^+ (a type of *closed-world assumption*). These are:

$$\{< A, B1 >, < A, B2 >, < A, C1 >, < A, C2 >, < B, C1 >, < B, C2 >,$$
$$< B, A1 >, < B, A2 >, < C, A1 >, < C, A2 >, < C, B1 >, < C, B2 >,$$
$$< B1, A >, < B2, A >, < C1, A >, < C2, A >, < C1, B >, < C2, B >,$$
$$< A1, B >, < A2, B >, < A1, C >, < A2, C >, < B1, C >, < B2, C >\}$$

There may be other cities not shown on this map, so the training set does not necessarily exhaust all the cities.

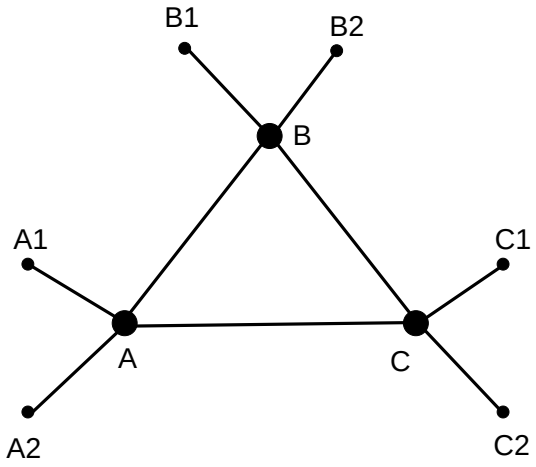


Figure 7.3: Part of an Airline Route Map

We want the learning program to induce a program for computing the value of the relation `Nonstop`. The training set, Ξ, can be thought of as a partial description of this relation in extensional form—it explicitly names some pairs in the relation and some pairs not in the relation. We desire to learn the `Nonstop` relation as a logic program in terms of the background relations, `Hub` and `Satellite`, which are also given in extensional form. Doing so will give us a more compact, *intensional*, description of the relation, and this description could well generalize usefully to other cities not mentioned in the map.

We assume the learning program has the following extensional definitions of the relations `Hub` and `Satellite`:

`Hub`

$$\{< A >, < B >, < C >\}$$

All other cities mentioned in the map are assumed not in the relation Hub. We will use the notation `Hub(x)` to express that the city named x is in the relation `Hub`.

`Satellite`

$$\{< A1, A, >, < A2, A >, < B1, B >, < B2, B >, < C1, C >, < C2, C >\}$$

All other pairs of cities mentioned in the map are not in the relation `Satellite`. We will use the notation `Satellite(x,y)` to express that the pair $< x, y >$ is in the relation `Satellite`.

Knowing that the predicate `Nonstop` is a two-place predicate, the inner loop of our algorithm initializes the first clause to `Nonstop(x,y) :-` . This clause is not necessary because it covers all the negative examples (since it covers all examples). So we must add a literal to its (empty) body. Suppose (selecting a literal from the refinement graph) the algorithm adds `Hub(x)`. The following positive instances in Ξ are covered by `Nonstop(x,y) :- Hub(x)`:

$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >,$
$< A, A1 >, < A, A2 >, < B, B1 >, < B, B2 >, < C, C1 >, < C, C2 >\}$

To compute this covering, we interpret the logic program `Nonstop(x,y) :- Hub(x)` for all pairs of cities in Ξ, using the pairs given in the background relation `Hub` as ground facts. The following negative instances are also covered:

$\{< A, B1 >, < A, B2 >, < A, C1 >, < A, C2 >, < C, A1 >, < C, A2 >,$
$< C, B1 >, < C, B2 >, < B, A1 >, < B, A2 >, < B, C1 >, < B, C2 >\}$

Thus, the clause is not yet necessary and another literal must be added. Suppose we next add `Hub(y)`. The following positive instances are covered by `Nonstop(x,y) :- Hub(x), Hub(y)`:

$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >\}$

There are no longer any negative instances in Ξ covered so the clause `Nonstop(x,y) :- Hub(x), Hub(y)` is necessary, and we can terminate the first pass through the inner loop.

But the program, π, consisting of just this clause is not sufficient. These positive instances are *not* covered by the clause:

$\{< A, A1 >, < A, A2 >, < A1, A >, < A2, A >, < B, B1 >, < B, B2 >,$
$< B1, B >, < B2, B >, < C, C1 >, < C, C2 >, < C1, C >, < C2, C >\}$

The positive instances that were covered by `Nonstop(x,y) :- Hub(x), Hub(y)` are removed from Ξ to form the Ξ_{cur} to be used in the next pass through the inner loop. Ξ_{cur} consists of all the negative instances in Ξ plus the positive instances (listed above) that are not yet covered. In order to attempt to cover them, the inner loop creates another clause c, initially set to `Nonstop(x,y) :-` . This clause covers all the negative instances, and so we must add literals to make it necessary. Suppose we add the literal `Satellite(x,y)`. The clause `Nonstop(x,y) :- Satellite(x,y)` covers no negative instances, so it is necessary. It does cover the following positive instances in Ξ_{cur}:

$\{< A1, A >, < A2, A >, < B1, B >, < B2, B >, < C1, C >, < C2, C >\}$

These instances are removed from Ξ_{cur} for the next pass through the inner loop. The program now contains two clauses:

```
Nonstop(x,y) :- Hub(x), Hub(y)
             :- Satellite(x,y)
```

This program is not yet sufficient since it does not cover the following positive instances:

$\{< A, A1 >, < A, A2 >, < B, B1 >, < B, B2 >, < C, C1 >, < C, C2 >\}$

During the next pass through the inner loop, we add the clause `Nonstop(x,y) :- Satellite(y,x)`. This clause is necessary, and since the program containing all three clauses is now sufficient, the procedure terminates with:

```
Nonstop(x,y) :- Hub(x), Hub(y)
             :- Satellite(x,y)
             :- Satellite(y,x)
```

Since each clause is necessary, and the whole program is sufficient, the program is also consistent with all instances of the training set. Note that this program can be applied (perhaps with good generalization) to other cities besides those in our partial map—so long as we can evaluate the relations `Hub` and `Satellite` for these other cities. In the next section, we show how the technique can be extended to use recursion on the relation we are inducing. With that extension, the method can be used to induce more general logic programs.

7.4 Inducing Recursive Programs

To induce a recursive program, we allow the addition of a literal having the same predicate letter as that in the head of the clause. Various mechanisms must be used to ensure that such a program will terminate; one such is to make sure that the new literal has different variables than those in the

head literal. The process is best illustrated with another example. Our example continues the one using the airline map, but we make the map somewhat simpler in order to reduce the size of the extensional relations used. Consider the map shown in Fig. 7.4. Again, B and C are hub cities, *B1* and *B2* are satellites of B, *C1* and *C2* are satellites of C. We have introduced two new cities, *B3* and *C3*. No flights exist between these cities and any other cities—perhaps there are only bus routes as shown by the grey lines in the map.

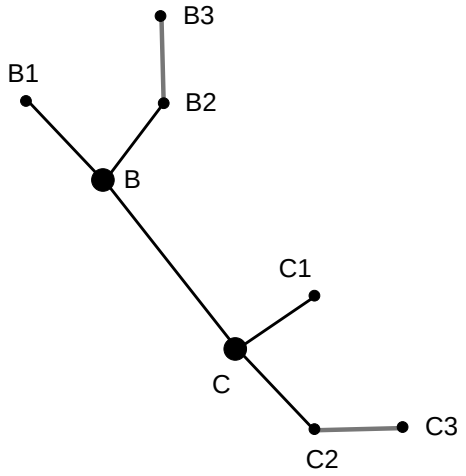


Figure 7.4: Another Airline Route Map

We now seek to learn a program for `Canfly(x,y)` that covers only those pairs of cities that can be reached by one or more nonstop flights. The relation `Canfly` is satisfied by the following pairs of postive instances:

$\{< B1, B >, < B1, B2 >, < B1, C >, < B1, C1 >, < B1, C2 >,$
$< B, B1 >, < B2, B1 >, < C, B1 >, < C1, B1 >, < C2, B1 >,$
$< B2, B >, < B2, C >, < B2, C1 >, < B2, C2 >, < B, B2 >,$
$< C, B2 >, < C1, B2 >, < C2, B2 >, < B, C >, < B, C1 >,$
$< B, C2 >, < C, B >, < C1, B >, < C2, B >, < C, C1 >,$
$< C, C2 >, < C1, C >, < C2, C >, < C1, C2 >, < C2, C1 >\}$

Using a closed-world assumption on our map, we take the negative instances of `Canfly` to be:

$\{< B3, B2 >, < B3, B >, < B3, B1 >, < B3, C >, < B3, C1 >,$
$< B3, C2 >, < B3, C3 >, < B2, B3 >, < B, B3 >, < B1, B3 >,$
$< C, B3 >, < C1, B3 >, < C2, B3 >, < C3, B3 >, < C3, B2 >,$
$< C3, B >, < C3, B1 >, < C3, C >, < C3, C1 >, < C3, C2 >,$
$< B2, C3 >, < B, C3 >, < B1, C3 >, < C, C3 >, < C1, C3 >,$
$< C2, C3 >\}$

We will induce `Canfly(x,y)` using the extensionally defined background relation `Nonstop` given earlier (modified as required for our reduced airline map) and `Canfly` itself (recursively).

As before, we start with the empty program and proceed to the inner loop to construct a clause that is necessary. Suppose that the inner loop adds the background literal `Nonstop(x,y)`. The clause `Canfly(x,y) :- Nonstop(x,y)` is necessary; it covers no negative instances. But it is not sufficient because it does not cover the following positive instances:

$\{< B1, B2 >, < B1, C >, < B1, C1 >, < B1, C2 >, < B2, B1 >,$
$< C, B1 >, < C1, B1 >, < C2, B1 >, < B2, C >, < B2, C1 >,$
$< B2, C2 >, < C, B2 >, < C1, B2 >, < C2, B2 >, < B, C1 >,$
$< B, C2 >, < C1, B >, < C2, B >, < C1, C2 >, < C2, C1 >\}$

Thus, we must add another clause to the program. In the inner loop, we first create the clause `Canfly(x,y) :- Nonstop(x,z)` which introduces the new variable z. We digress briefly to describe how a program containing a clause with unbound variables in its body is interpreted. Suppose we try to interpret it for the positive instance `Canfly(B1,B2)`. The interpreter attempts to establish `Nonstop(B1,z)` for some z. Since `Nonstop(B1, B)`, for example, is a background fact, the interpreter returns T—which means that the instance $< B1, B2 >$ is covered. Suppose now, we attempt to interpret the clause for the negative instance `Canfly(B3,B)`. The interpreter attempts to establish `Nonstop(B3,z)` for some z. There are no background facts that match, so the clause does not cover $< B3, B >$. Using the interpreter, we see that the clause `Canfly(x,y) :- Nonstop(x,z)` covers all of the positive instances not already covered by the first clause, but it also covers many negative instances such as $< B2, B3 >$, and $< B, B3 >$. So the inner

loop must add another literal. This time, suppose it adds `Canfly(z,y)` to yield the clause `Canfly(x,y) :- Nonstop(x,z), Canfly(z,y)`. This clause is necessary; no negative instances are covered. The program is now sufficient and consistent; it is:

```
Canfly(x,y) :- Nonstop(x,y)
            :- Nonstop(x,z), Canfly(z,y)
```

7.5 Choosing Literals to Add

One of the first practical ILP systems was Quinlan's FOIL [Quinlan, 1990]. A major problem involves deciding how to select a literal to add in the inner loop (from among the literals that are allowed). In FOIL, Quinlan suggested that candidate literals can be compared using an information-like measure—similar to the measures used in inducing decision trees. A measure that gives the same comparison as does Quinlan's is based on the amount by which adding a literal increases the *odds* that an instance drawn at random from those covered by the new clause is a positive instance beyond what these odds were before adding the literal.

Let p be an estimate of the probability that an instance drawn at random from those covered by a clause before adding the literal is a positive instance. That is, p =(number of positive instances covered by the clause)/(total number of instances covered by the clause). It is convenient to express this probability in "odds form." The odds, o, that a covered instance is positive is defined to be $o = p/(1-p)$. Expressing the probability in terms of the odds, we obtain $p = o/(1+o)$.

After selecting a literal, l, to add to a clause, some of the instances previously covered are still covered; some of these are positive and some are negative. Let p_l denote the probability that an instance drawn at random from the instances covered by the new clause (with l added) is positive. The odds will be denoted by o_l. We want to select a literal, l, that gives maximal increase in these odds. That is, if we define $\lambda_l = o_l/o$, we want a literal that gives a high value of λ_l. Specializing the clause in such a way that it fails to cover many of the negative instances previously covered but still covers most of the positive instances previously covered will result in a high value of λ_l. (It turns out that the value of Quinlan's information theoretic measure increases monotonically with λ_l, so we could just as well use the latter instead.)

Besides finding a literal with a high value of λ_l, Quinlan's FOIL system also restricts the choice to literals that:

a) contain at least one variable that has already been used,

b) place further restrictions on the variables if the literal selected has the same predicate letter as the literal being induced (in order to prevent infinite recursion), and

c) survive a pruning test based on the values of λ_l for those literals selected so far.

We refer the reader to Quinlan's paper for further discussion of these points. Quinlan also discusses post-processing pruning methods and presents experimental results of the method applied to learning recursive relations on lists, on learning rules for chess endgames and for the card game Eleusis, and for some other standard tasks mentioned in the machine learning literature.

The reader should also refer to [Pazzani & Kibler, 1992, Lavrač & Džeroski, 1994, Muggleton, 1991, Muggleton, 1992].

Discuss preprocessing, postprocessing, bottom-up methods, and LINUS.

7.6 Relationships Between ILP and Decision Tree Induction

The generic ILP algorithm can also be understood as a type of decision tree induction. Recall the problem of inducing decision trees when the values of attributes are categorical. When splitting on a single variable, the split at each node involves asking to which of several mutually exclusive and exhaustive subsets the value of a variable belongs. For example, if a node tested the variable x_i, and if x_i could have values drawn from $\{A, B, C, D, E, F\}$, then one possible split (among many) might be according to whether the value of x_i had as value one of $\{A, B, C\}$ or one of $\{D, E, F\}$.

It is also possible to make a multi-variate split—testing the values of two or more variables at a time. With categorical variables, an n-variable split would be based on which of several n-ary relations the values of the variables satisfied. For example, if a node tested the variables x_i and x_j, and if x_i and x_j both could have values drawn from $\{A, B, C, D, E, F\}$, then one possible binary split (among many) might be according to whether or not $< x_i, x_j >$ satisfied the relation $\{< A, C >, < C, D >\}$. (Note that our subset method of forming single-variable splits could equivalently have been framed using 1-ary relations—which are usually called properties.)

In this framework, the ILP problem is as follows: We are given a training set, Ξ, of positively and negatively labeled patterns whose components are

drawn from a set of variables $\{x, y, z, \ldots\}$. The positively labeled patterns in Ξ form an extensional definition of a relation, R. We are also given background relations, $R_1, \ldots, R_k$, on various subsets of these variables. (That is, we are given sets of tuples that are in these relations.) We desire to construct an intensional definition of R in terms of the $R_1, \ldots, R_k$, such that all of the positively labeled patterns in Ξ are satisfied by R and none of the negatively labeled patterns are. The intensional definition will be in terms of a logic program in which the relation R is the head of a set of clauses whose bodies involve the background relations.

The generic ILP algorithm can be understood as decision tree induction, where each node of the decision tree is itself a sub-decision tree, and each sub-decision tree consists of nodes that make binary splits on several variables using the background relations, R_i. Thus we will speak of a top-level decision tree and various sub-decision trees. (Actually, our decision trees will be decision lists—a special case of decision trees, but we will refer to them as trees in our discussions.)

In broad outline, the method for inducing an intensional version of the relation R is illustrated by considering the decision tree shown in Fig. 7.5. In this diagram, the patterns in Ξ are first filtered through the decision tree in top-level node 1. The background relation R_1 is satisfied by some of these patterns; these are filtered to the right (to relation R_2), and the rest are filtered to the left (more on what happens to these later). Right-going patterns are filtered through a sequence of relational tests until only positively labeled patterns satisfy the last relation—in this case R_3. That is, the subset of patterns satisfying all the relations, R_1, R_2, and R_3 contains only positive instances from Ξ. (We might say that this combination of tests is necessary. They correspond to the clause created in the first pass through the inner loop of the generic ILP algorithm.) Let us call the subset of patterns satisfying these relations, Ξ_1; these satisfy Node 1 at the top level. All other patterns, that is $\{\Xi - \Xi_1\} = \Xi_2$ are filtered to the left by Node 1.

Ξ_2 is then filtered by top-level Node 2 in much the same manner, so that Node 2 is satisfied only by the positively labeled samples in Ξ_2. We continue filtering through top-level nodes until only the negatively labeled patterns fail to satisfy a top node. In our example, Ξ_4 contains only negatively labeled patterns and the union of Ξ_1 and Ξ_3 contains all the positively labeled patterns. The relation, R, that distinguishes positive from negative patterns in Ξ is then given in terms of the following logic program:

```
R :- R1, R2, R3
  :- R4, R5
```

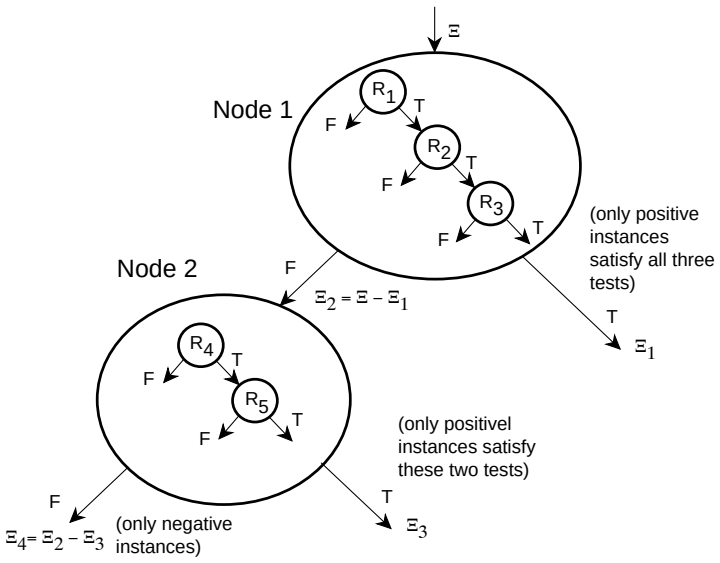


Figure 7.5: A Decision Tree for ILP

If we apply this sort of decision-tree induction procedure to the problem of generating a logic program for the relation `Nonstop` (refer to Fig. 7.3), we obtain the decision tree shown in Fig. 7.6. The logic program resulting from this decision tree is the same as that produced by the generic ILP algorithm.

In setting up the problem, the training set, Ξ can be expressed as a set of 2-dimensional vectors with components x and y. The values of these components range over the cities $\{A, B, C, A1, A2, B1, B2, C1, C2\}$ except (for simplicity) we do not allow patterns in which x and y have the same value. As before, the relation, `Nonstop`, contains the following pairs of cities, which are the positive instances:

$$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >,$$
$$< A, A1 >, < A, A2 >, < A1, A >, < A2, A >, < B, B1 >, < B, B2 >,$$
$$< B1, B >, < B2, B >, < C, C1 >, < C, C2 >, < C1, C >, < C2, C >\}$$

All other pairs of cities named in the map of Fig. 7.3 (using the closed world assumption) are not in the relation `Nonstop` and thus are negative instances.

Because the values of x and y are categorical, decision-tree induction would be a very difficult task—involving as it does the need to invent relations on x and y to be used as tests. But with the background relations, R_i (in this case `Hub` and `Satellite`), the problem is made much easier. We select these relations in the same way that we select literals; from among the available tests, we make a selection based on which leads to the largest value of λ_{R_i}.

7.7 Bibliographical and Historical Remarks

To be added.

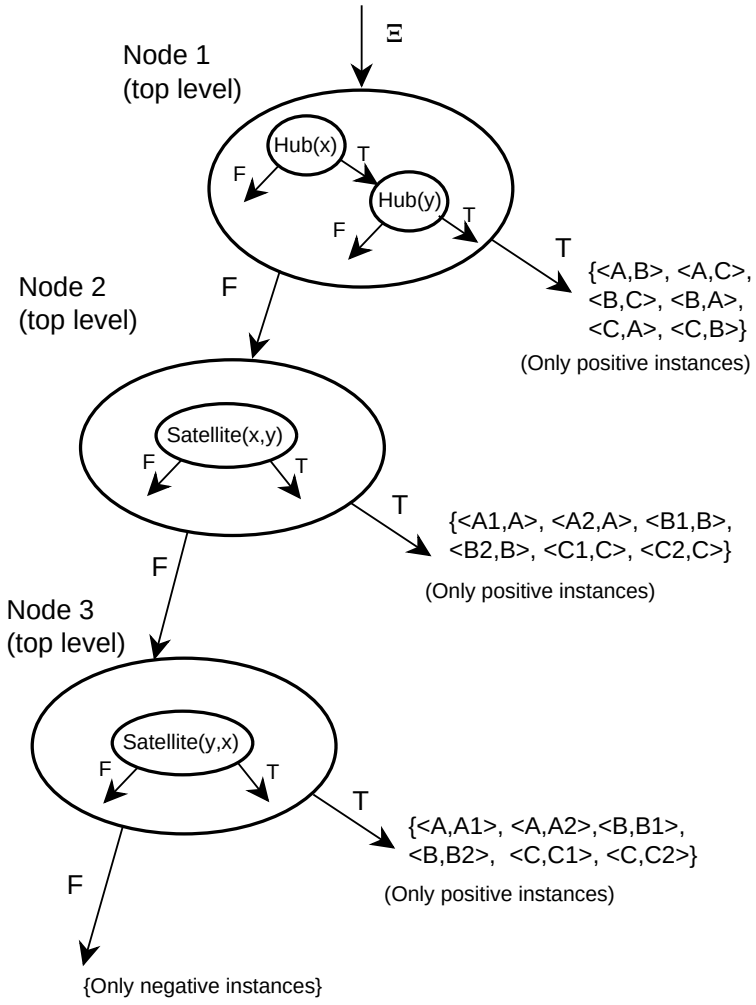


Figure 7.6: A Decision Tree for the Airline Route Problem

Chapter 8

Computational Learning Theory

In chapter one we posed the problem of guessing a function given a set of sample inputs and their values. We gave some intuitive arguments to support the claim that after seeing only a small fraction of the possible inputs (and their values) that we could guess *almost correctly* the values of *most* subsequent inputs—if we knew that the function we were trying to guess belonged to an appropriately restricted subset of functions. That is, a given training set of sample patterns might be adequate to allow us to select a function, *consistent with the labeled samples*, from among a restricted set of hypotheses such that with high *probability* the function we select will be *approximately correct* (small probability of error) on subsequent samples drawn at random according to the *same distribution* from which the labeled samples were drawn. This insight led to the theory of *probably approximately correct (PAC)* learning—initially developed by Leslie Valiant [Valiant, 1984]. We present here a brief description of the theory for the case of Boolean functions. [Dietterich, 1990, Haussler, 1988, Haussler, 1990] give nice surveys of the important results.

Other overviews?

8.1 Notation and Assumptions for PAC Learning Theory

We assume a training set Ξ of n-dimensional vectors, $\mathbf{X}_i$, $i = 1, \ldots, m$, each labeled (by 1 or 0) according to a target function, f, which is unknown to

the learner. The probability of any given vector $\mathbf{X}$ being in Ξ, or later being presented to the learner, is $P(\mathbf{X})$. The probability distribution, P, can be arbitrary. (In the literature of PAC learning theory, the target function is usually called the target *concept* and is denoted by c, but to be consistent with our previous notation we will continue to denote it by f.) Our problem is to guess a function, $h(\mathbf{X})$, based on the labeled samples in Ξ. In PAC theory such a guessed function is called the *hypothesis*. We assume that the target function is some element of a set of functions, $\mathcal{C}$. We also assume that the hypothesis, h, is an element of a set, $\mathcal{H}$, of hypotheses, which includes the set, $\mathcal{C}$, of target functions. $\mathcal{H}$ is called the *hypothesis space*.

In general, h won't be identical to f, but we can strive to have the value of $h(\mathbf{X})$ = the value of $f(\mathbf{X})$ for *most* $\mathbf{X}$'s. That is, we want h to be *approximately* correct. To quantify this notion, we define the *error* of h, ε_h, as the probability that an $\mathbf{X}$ drawn randomly according to P will be misclassified:

$$\varepsilon_h = \sum_{[\mathbf{X}: h(\mathbf{X}) \neq f(\mathbf{X})]} P(\mathbf{X})$$

Boldface symbols need to be smaller when they are subscripts in math environments.

We say that h is *approximately* (except for ε) *correct* if $\varepsilon_h \leq \varepsilon$, where ε is the *accuracy parameter*.

Suppose we are able to find an h that classifies *all* m randomly drawn training samples correctly; that is, h is consistent with this randomly selected training set, Ξ. If m is large enough, will such an h be approximately correct (and for what value of ε)? On some training occasions, using m randomly drawn training samples, such an h might turn out to be approximately correct (for a given value of ε), and on others it might not. We say that h is *probably* (except for δ) *approximately correct (PAC)* if the probability that it is approximately correct is greater than $1 - \delta$, where δ is the *confidence parameter*. We shall show that if m is greater than some bound whose value depends on ε and δ, such an h is guaranteed to be probably approximately correct.

In general, we say that a learning algorithm *PAC-learns* functions from $\mathcal{C}$ in terms of $\mathcal{H}$ iff for every function $f \epsilon\ \mathcal{C}$, it outputs a hypothesis $h \epsilon\ \mathcal{H}$, such that with probability at least $(1 - \delta)$, $\varepsilon_h \leq \varepsilon$. Such a hypothesis is called *probably* (except for δ) *approximately* (except for ε) *correct*.

We want learning algorithms that are tractable, so we want an algorithm that PAC-learns functions in polynomial time. This can only be done for certain classes of functions. If there are a finite number of hypotheses in a hypothesis set (as there are for many of the hypothesis sets we have considered), we could always produce a consistent hypothesis from this

set by testing all of them against the training data. But if there are an exponential number of hypotheses, that would take exponential time. We seek training methods that produce consistent hypotheses in less time. The time complexities for various hypothesis sets have been determined, and these are summarized in a table to be presented later.

A class, $\mathcal{C}$, is *polynomially PAC learnable* in terms of $\mathcal{H}$ provided there exists a polynomial-time learning algorithm (polynomial in the number of samples needed, m, in the dimension, n, in $1/\varepsilon$, and in $1/\delta$) that PAC-learns functions in $\mathcal{C}$ in terms of $\mathcal{H}$.

Initial work on PAC assumed $\mathcal{H} = \mathcal{C}$, but it was later shown that some functions cannot be polynomially PAC-learned under such an assumption (assuming P $\neq$ NP)—but *can* be polynomially PAC-learned if $\mathcal{H}$ is a strict superset of $\mathcal{C}$! Also our definition does not specify the distribution, P, from which patterns are drawn nor does it say anything about the properties of the learning algorithm. Since $\mathcal{C}$ and $\mathcal{H}$ do not have to be identical, we have the further restrictive definition:

A *properly PAC-learnable* class is a class $\mathcal{C}$ for which there exists an algorithm that polynomially PAC-learns functions from $\mathcal{C}$ in terms of $\mathcal{C}$.

8.2 PAC Learning

8.2.1 The Fundamental Theorem

Suppose our learning algorithm selects some h randomly from among those that are consistent with the values of f on the m training patterns. The probability that the error of this randomly selected h is *greater* than some ε, with h consistent with the values of $f(\mathbf{X})$ for m instances of $\mathbf{X}$ (drawn according to arbitrary P), is less than or equal to $|\mathcal{H}|e^{-\varepsilon m}$, where $|\mathcal{H}|$ is the number of hypotheses in $\mathcal{H}$. We state this result as a theorem [Blumer, *et al.*, 1987]:

Theorem 8.1 (Blumer, *et al.***)** *Let $\mathcal{H}$ be any set of hypotheses, Ξ be a set of $m \geq 1$ training examples drawn independently according to some distribution P, f be any classification function in $\mathcal{H}$, and $\varepsilon > 0$. Then, the probability that there exists a hypothesis h consistent with f for the members of Ξ but with error greater than ε is at most $|\mathcal{H}|e^{-\varepsilon m}$.*

Proof:

Consider the set of all hypotheses, $\{h_1, h_2, \ldots, h_i, \ldots, h_S\}$, in $\mathcal{H}$, where $S = |\mathcal{H}|$. The error for h_i is ε_{h_i}= the probability that h_i will classify a pattern in error (that is, differently than f would classify it). The probability

that h_i will classify a pattern correctly is $(1-\varepsilon_{h_i})$. A subset, $\mathcal{H}_B$, of $\mathcal{H}$ will have error greater than ε. We will call the hypotheses in this subset *bad*. The probability that any particular one of these bad hypotheses, say h_b, would classify a pattern correctly is $(1-\varepsilon_{h_b})$. Since $\varepsilon_{h_b} > \varepsilon$, the probability that h_b (or any other bad hypothesis) would classify a pattern correctly is less than $(1-\varepsilon)$. The probability that it would classify *all* m independently drawn patterns correctly is then less than $(1-\varepsilon)^m$.

That is,

prob[h_b classifies all m patterns correctly $|h_b \in \mathcal{H}_B] \leq (1-\varepsilon)^m$.

prob[*some* $h \in \mathcal{H}_B$ classifies all m patterns correctly]
$= \sum_{h_b \in \mathcal{H}_B}$ prob[h_b classifies all m patterns correctly $|h_b \in \mathcal{H}_B$]
$\leq K(1-\varepsilon)^m$, where $K = |\mathcal{H}_B|$.

That is,

prob[there is a bad hypothesis that classifies all m patterns correctly]
$\leq K(1-\varepsilon)^m$.

Since $K \leq |\mathcal{H}|$ and $(1-\varepsilon)^m \leq e^{-\varepsilon m}$, we have:

prob[there is a bad hypothesis that classifies all m patterns correctly]
= prob[there is a hypothesis with error $> \varepsilon$ and that classifies all m patterns correctly] $\leq |\mathcal{H}|e^{-\varepsilon m}$.

□

A corollary of this theorem is:

Corollary 8.2 *Given $m \geq (1/\varepsilon)(\ln|\mathcal{H}| + \ln(1/\delta))$ independent samples, the probability that there exists a hypothesis in $\mathcal{H}$ that is consistent with f on these samples and has error greater than ε is at most δ.*

Proof: We are to find a bound on m that guarantees that

prob[there is a hypothesis with error $> \varepsilon$ and that classifies all m patterns correctly] $\leq \delta$. Thus, using the result of the theorem, we must show that $|\mathcal{H}|e^{-\varepsilon m} \leq \delta$. Taking the natural logarithm of both sides yields:

$$\ln|\mathcal{H}| - \varepsilon m \leq \ln\delta$$

or

$$m \geq (1/\varepsilon)(\ln |\mathcal{H}| + \ln(1/\delta))$$

□

This corollary is important for two reasons. First it clearly states that we can select *any* hypothesis consistent with the m samples and be assured that with probability $(1 - \delta)$ its error will be less than ε. Also, it shows that in order for m to increase no more than polynomially with n, $|\mathcal{H}|$ can be no larger than $2^{O(n^k)}$. No class larger than that can be guaranteed to be properly PAC learnable.

Here is a possible point of confusion: The bound given in the corollary is an *upper bound* on the value of m needed to guarantee polynomial probably approximately correct learning. Values of m greater than that bound are sufficient (but might not be necessary). We will present a lower (necessary) bound later in the chapter.

8.2.2 Examples

Terms

Let $\mathcal{H}$ be the set of terms (conjunctions of literals). Then, $|\mathcal{H}| = 3^n$, and

$$m \geq (1/\varepsilon)(\ln(3^n) + \ln(1/\delta))$$

$$\geq (1/\varepsilon)(1.1n + \ln(1/\delta))$$

Note that the bound on m increases only polynomially with n, $1/\varepsilon$, and $1/\delta$.

For $n = 50$, $\varepsilon = 0.01$ and $\delta = 0.01$, $m \geq 5,961$ guarantees PAC learnability.

In order to show that terms are *properly PAC learnable*, we additionally have to show that one can find in time polynomial in m and n a hypothesis h consistent with a set of m patterns labeled by the value of a term. The following procedure for finding such a consistent hypothesis requires $O(nm)$ steps (adapted from [Dietterich, 1990, page 268]):

We are given a training sequence, Ξ, of m examples. Find the first pattern, say $\mathbf{X}_1$, in that list that is labeled with a 1. Initialize a Boolean function, h, to the conjunction of the n literals corresponding to the values

of the n components of $\mathbf{X}_1$. (Components with value 1 will have corresponding positive literals; components with value 0 will have corresponding negative literals.) If there are no patterns labeled by a 1, we exit with the null concept ($h \equiv 0$ for all patterns). Then, for each additional pattern, $\mathbf{X}_i$, that is labeled with a 1, we delete from h any Boolean variables appearing in $\mathbf{X}_i$ with a sign different from their sign in h. After processing all the patterns labeled with a 1, we check all of the patterns labeled with a 0 to make sure that none of them is assigned value 1 by h. If, at any stage of the algorithm, any patterns labeled with a 0 are assigned a 1 by h, then there exists no term that consistently classifies the patterns in Ξ, and we exit with failure. Otherwise, we exit with h.

Change this paragraph if this algorithm was presented in Chapter Three.

As an example, consider the following patterns, all labeled with a 1 (from [Dietterich, 1990]):

$(0,1,1,0)$

$(1,1,1,0)$

$(1,1,0,0)$

After processing the first pattern, we have $h = \overline{x_1}x_2x_3\overline{x_4}$; after processing the second pattern, we have $h = x_2x_3\overline{x_4}$; finally, after the third pattern, we have $h = x_2\overline{x_4}$.

Linearly Separable Functions

Let $\mathcal{H}$ be the set of all linearly separable functions. Then, $|\mathcal{H}| \leq 2^{n^2}$, and

$$m \geq (1/\varepsilon)\big(n^2 \ln 2 + \ln(1/\delta)\big)$$

Again, note that the bound on m increases only polynomially with n, $1/\varepsilon$, and $1/\delta$.

For $n = 50$, $\varepsilon = 0.01$ and $\delta = 0.01$, $m \geq 173,748$ guarantees PAC learnability.

To show that linearly separable functions are *properly PAC learnable*, we would have additionally to show that one can find in time polynomial in m and n a hypothesis h consistent with a set of m labeled linearly separable patterns.

Linear programming is polynomial.

8.2.3 Some Properly PAC-Learnable Classes

Some properly PAC-learnable classes of functions are given in the following table. (Adapted from [Dietterich, 1990, pages 262 and 268] which also gives references to proofs of some of the time complexities.)

$\mathcal{H}$	$\lvert\mathcal{H}\rvert$	Time Complexity	P. Learnable?
terms	3^n	polynomial	yes
k-term DNF (k disjunctive terms)	$2^{O(kn)}$	NP-hard	no
k-DNF (a disjunction of k-sized terms)	$2^{O(n^k)}$	polynomial	yes
k-CNF (a conjunction of k-sized clauses)	$2^{O(n^k)}$	polynomial	yes
k-DL (decision lists with k-sized terms)	$2^{O(n^k k \lg n)}$	polynomial	yes
lin. sep.	$2^{O(n^2)}$	polynomial	yes
lin. sep. with (0,1) weights	?	NP-hard	no
k-2NN	?	NP-hard	no
DNF (all Boolean functions)	2^{2^n}	polynomial	no

(Members of the class k-2NN are two-layer, feedforward neural networks with exactly k hidden units and one output unit.)

Summary: In order to show that a class of functions is *Properly PAC-Learnable* :

a. Show that there is an algorithm that produces a consistent hypothesis on m n-dimensional samples in time polynomial in m and n.

b. Show that the sample size, m, needed to ensure PAC learnability is polynomial (or better) in $(1/\varepsilon)$, $(1/\delta)$, and n by showing that $\ln \lvert\mathcal{H}\rvert$ is polynomial or better in the number of dimensions.

As hinted earlier, sometimes enlarging the class of hypotheses makes learning easier. For example, the table above shows that k-CNF is PAC learnable, but k-term-DNF is not. And yet, k-term-DNF is a subclass of k-CNF! So, even if the target function were in k-term-DNF, one would be able to find a hypothesis in k-CNF that is probably approximately correct for the target function. Similarly, linearly separable functions implemented by TLUs whose weight values are restricted to 0 and 1 are not properly PAC learnable, whereas unrestricted linearly separable functions are. It is possible that enlarging the space of hypotheses makes finding one that is consistent with the training examples easier. An interesting question is whether or not the class of functions in k-2NN is polynomially PAC

learnable if the hypotheses are drawn from k'-2NN with $k' > k$. (At the time of writing, this matter is still undecided.)

Although PAC learning theory is a powerful analytic tool, it (like complexity theory) deals mainly with worst-case results. The fact that the class of two-layer, feedforward neural networks is not polynomially PAC learnable is more an attack on the theory than it is on the networks, which have had many successful applications. As [Baum, 1994, page 416-17] says: " ... humans are capable of learning in the natural world. Therefore, a proof within some model of learning that learning is not feasible is an indictment of the model. We should examine the model to see what constraints can be relaxed and made more realistic."

8.3 The Vapnik-Chervonenkis Dimension

8.3.1 Linear Dichotomies

Consider a set, $\mathcal{H}$, of functions, and a set, Ξ, of (unlabeled) patterns. One measure of the expressive power of a set of hypotheses, relative to Ξ, is its ability to make *arbitrary* classifications of the patterns in Ξ.[1] If there are m patterns in Ξ, there are 2^m different ways to divide these patterns into two disjoint and exhaustive subsets. We say there are 2^m different *dichotomies* of Ξ. If Ξ were to include *all* of the 2^n Boolean patterns, for example, there are 2^{2^n} ways to dichotomize them, and (of course) the set of all possible Boolean functions dichotomizes them in all of these ways. But a subset, $\mathcal{H}$, of the Boolean functions might not be able to dichotomize an arbitrary set, Ξ, of m Boolean patterns in all 2^m ways. In general (that is, even in the non-Boolean case), we say that if a subset, $\mathcal{H}$, of functions can dichotomize a set, Ξ, of m patterns in all 2^m ways, then $\mathcal{H}$ *shatters* Ξ.

As an example, consider a set Ξ of m patterns in the n-dimensional space, $\mathcal{R}^n$. (That is, the n components of these patterns are real numbers.) We define a *linear dichotomy* as one implemented by an $(n-1)$-dimensional hyperplane in the n-dimensional space. How many linear dichotomies of m patterns in n dimensions are there? For example, as shown in Fig. 8.1, there are 14 dichotomies of four points in two dimensions (each separating line yields two dichotomies depending on whether the points on one side of the line are classified as 1 or 0). (Note that even though there are an infinite number of hyperplanes, there are, nevertheless, only a finite number

[1] And, of course, if a hypothesis drawn from a set that could make arbitrary classifications of a set of training patterns, there is little likelihood that such a hypothesis will generalize well beyond the training set.

of ways in which hyperplanes can dichotomize a finite number of patterns. Small movements of a hyperplane typically do not change the classifications of any patterns.)

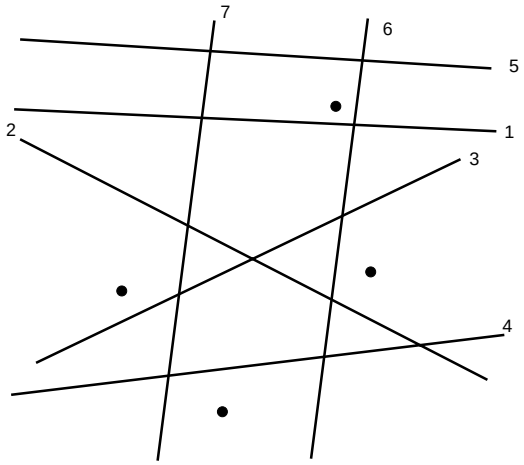


Figure 8.1: Dichotomizing Points in Two Dimensions

The number of dichotomies achievable by hyperplanes depends on how the patterns are disposed. For the maximum number of linear dichotomies, the points must be in what is called *general position*. For $m > n$, we say that a set of m points is in *general position* in an n-dimensional space if and only if no subset of $(n+1)$ points lies on an $(n-1)$-dimensional hyperplane. When $m \leq n$, a set of m points is in general position if no $(m-2)$-dimensional hyperplane contains the set. Thus, for example, a set of $m \geq 4$ points is in general position in a three-dimensional space if no four of them lie on a (two-dimensional) plane. We will denote the number of linear dichotomies of m points in general position in an n-dimensional space by the expression $\Pi_L(m,n)$.

It is not too difficult to verify that:

Include the derivation.

$$\Pi_L(m,n) = 2\sum_{i=0}^{n} C(m-1,i) \qquad \text{for } m > n, \text{and}$$

$$= 2^m \qquad \text{for } m \leq n$$

where $C(m-1,i)$ is the binomial coefficient $\frac{(m-1)!}{(m-1-i)!i!}$.

The table below shows some values for $\Pi_L(m,n)$.

m (no. of patterns)	n (dimension)				
	1	2	3	4	5
1	2	2	2	2	2
2	**4**	4	4	4	4
3	6	**8**	8	8	8
4	8	14	**16**	16	16
5	10	22	30	**32**	32
6	12	32	52	62	**64**
7	14	44	84	114	126
8	16	58	128	198	240

Note that the class of linear dichotomies shatters the m patterns if $m \leq n+1$. The bold-face entries in the table correspond to the highest values of m for which linear dichotomies shatter m patterns in n dimensions.

8.3.2 Capacity

Let $P_{m,n} = \frac{\Pi_L(m,n)}{2^m}$ = the probability that a randomly selected dichotomy (out of the 2^m possible dichotomies of m patterns in n dimensions) will be linearly separable. In Fig. 8.2 we plot $P_{\lambda(n+1),n}$ versus λ and n, where $\lambda = m/(n+1)$.

Note that for large n (say $n > 30$) how quickly $P_{m,n}$ falls from 1 to 0 as m goes above $2(n+1)$. For $m < 2(n+1)$, *any* dichotomy of the m points is almost certainly linearly separable. But for $m > 2(n+1)$, a randomly selected dichotomy of the m points is almost certainly not linearly separable. For this reason $m = 2(n+1)$ is called the *capacity* of a TLU [Cover, 1965]. Unless the number of training patterns exceeds the capacity, the fact that a TLU separates those training patterns according to their labels means nothing in terms of how well that TLU will generalize to new patterns. There is nothing special about a separation found for $m < 2(n+1)$ patterns—almost *any* dichotomy of those patterns would have been linearly separable. To make sure that the separation found is forced by the training set and thus generalizes well, it has to be the case that there are very few linearly separable functions that would separate the m training patterns.

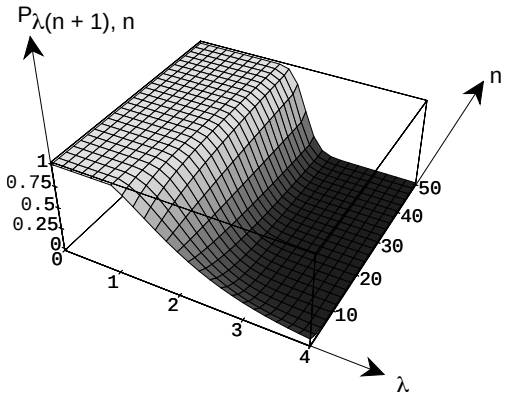


Figure 8.2: Probability that a Random Dichotomy is Linearly Separable

Analogous results about the generalizing abilities of neural networks have been developed by [Baum & Haussler, 1989] and given intuitive and experimental justification in [Baum, 1994, page 438]:

> "The results seemed to indicate the following heuristic rule holds. If M examples [can be correctly classified by] a net with W weights (for $M >> W$), the net will make a fraction ε of errors on new examples chosen from the same [uniform] distribution where $\varepsilon = W/M$."

8.3.3 A More General Capacity Result

Corollary 7.2 gave us an expression for the number of training patterns sufficient to guarantee a required level of generalization—assuming that the function we were guessing was a function belonging to a class of known and finite cardinality. The capacity result just presented applies to linearly separable functions for non-binary patterns. We can extend these ideas to general dichotomies of non-binary patterns.

In general, let us denote the maximum number of dichotomies of *any* set of m n-dimensional patterns by hypotheses in $\mathcal{H}$ as $\Pi_{\mathcal{H}}(m, n)$. The number of dichotomies will, of course, depend on the disposition of the m points

in the n-dimensional space; we take $\Pi_{\mathcal{H}}(m, n)$ to be the maximum over all possible arrangements of the m points. (In the case of the class of linearly separable functions, the maximum number is achieved when the m points are in general position.) For each class, $\mathcal{H}$, there will be some maximum value of m for which $\Pi_{\mathcal{H}}(m, n) = 2^m$, that is, for which $\mathcal{H}$ shatters the m patterns. This maximum number is called the *Vapnik-Chervonenkis (VC) dimension* and is denoted by VCdim($\mathcal{H}$) [Vapnik & Chervonenkis, 1971].

We saw that for the class of linear dichotomies, VCdim($Linear$) = (n+ 1). As another example, let us calculate the VC dimension of the hypothesis space of single intervals on the real line—used to classify points on the real line. We show an example of how points on the line might be dichotomized by a single interval in Fig. 8.3. The set Ξ could be, for example, {0.5, 2.5, - 2.3, 3.14}, and one of the hypotheses in our set would be [1, 4.5]. This hypothesis would label the points 2.5 and 3.14 with a 1 and the points - 2.3 and 0.5 with a 0. This set of hypotheses (single intervals on the real line) can arbitrarily classify any two points. But no single interval can classify three points such that the outer two are classified as 1 and the inner one as 0. Therefore the VC dimension of single intervals on the real line is 2. As soon as we have many more than 2 training patterns on the real line and provided we know that the classification function we are trying to guess is a single interval, then we begin to have good generalization.

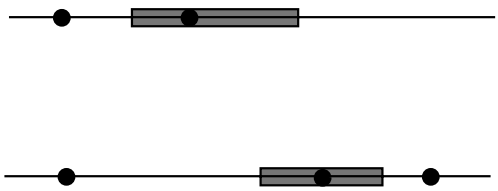


Figure 8.3: Dichotomizing Points by an Interval

The VC dimension is a useful measure of the expressive power of a hypothesis set. Since *any* dichotomy of VCdim($\mathcal{H}$) or fewer patterns in general position in n dimensions can be achieved by *some* hypothesis in $\mathcal{H}$, we must have many more than VCdim($\mathcal{H}$) patterns in the training set in order that a hypothesis consistent with the training set is sufficiently constrained to imply good generalization. Our examples have shown that the concept of VC dimension is not restricted to Boolean functions.

8.3.4 Some Facts and Speculations About the VC Dimension

- If there are a finite number, $|\mathcal{H}|$, of hypotheses in $\mathcal{H}$, then:
  $\text{VCdim}(\mathcal{H}) \leq \log(|\mathcal{H}|)$
- The VC dimension of terms in n dimensions is n.
- Suppose we generalize our example that used a hypothesis set of single intervals on the real line. Now let us consider an n-dimensional feature space and tests of the form $L_i \leq x_i \leq H_i$. We allow only one such test per dimension. A hypothesis space consisting of conjunctions of these tests (called *axis-parallel hyper-rectangles*) has VC dimension bounded by:
  $n \leq \text{ VCdim } \leq 2n$
- As we have already seen, TLUs with n inputs have a VC dimension of $n + 1$.
- [Baum, 1994, page 438] gives experimental evidence for the proposition that " ... multilayer [neural] nets have a VC dimension roughly equal to their total number of [adjustable] weights."

8.4 VC Dimension and PAC Learning

There are two theorems that connect the idea of VC dimension with PAC learning [Blumer, *et al.*, 1990]. We state these here without proof.

Theorem 8.3 (Blumer, *et al.*) *A hypothesis space $\mathcal{H}$ is PAC learnable iff it has finite VC dimension.*

Theorem 8.4 *A set of hypotheses, $\mathcal{H}$, is properly PAC learnable if:*

a. $m \geq (1/\varepsilon)\max[4\lg(2/\delta),\ 8\ \text{VCdim}\ \lg(13/\varepsilon)]$, *and*

b. if there is an algorithm that outputs a hypothesis $h \in \mathcal{H}$ consistent with the training set in polynomial (in m and n) time.

The second of these two theorems improves the bound on the number of training patterns needed for linearly separable functions to one that is linear in n. In our previous example of how many training patterns were needed to ensure PAC learnability of a linearly separable function if $n = 50$,

$\varepsilon = 0.01$, and $\delta = 0.01$, we obtained $m \geq 173,748$. Using the Blumer, *et al.* result we would get $m \geq 52,756$.

As another example of the second theorem, let us take $\mathcal{H}$ to be the set of closed intervals on the real line. The VC dimension is 2 (as shown previously). With $n = 50$, $\varepsilon = 0.01$, and $\delta = 0.01$, $m \geq 16,551$ ensures PAC learnability.

There is also a theorem that gives a lower (necessary) bound on the number of training patterns required for PAC learning [Ehrenfeucht, *et al.*, 1988]:

Theorem 8.5 *Any PAC learning algorithm must examine at least* $\Omega(1/\varepsilon \lg(1/\delta) + \text{VCdim}(\mathcal{H}))$ *training patterns.*

The difference between the lower and upper bounds is $O(\log(1/\varepsilon)\text{VCdim}(\mathcal{H})/\varepsilon)$.

8.5 Bibliographical and Historical Remarks

To be added.

Chapter 9

Unsupervised Learning

9.1 What is Unsupervised Learning?

Consider the various sets of points in a two-dimensional space illustrated in Fig. 9.1. The first set (a) seems naturally partitionable into two classes, while the second (b) seems difficult to partition at all, and the third (c) is problematic. *Unsupervised learning* uses procedures that attempt to find natural partitions of patterns. There are two stages:

- Form an R-way partition of a set Ξ of *unlabeled* training patterns (where the value of R, itself, may need to be induced from the patterns). The partition separates Ξ into R mutually exclusive and exhaustive subsets, $\Xi_1, \ldots, \Xi_R$, called *clusters*.
- Design a classifier based on the labels assigned to the training patterns by the partition.

We will explain shortly various methods for deciding how many clusters there should be and for separating a set of patterns into that many clusters. We can base some of these methods, and their motivation, on minimum-description-length (MDL) principles. In that setting, we assume that we want to encode a description of a set of points, Ξ, into a message of minimal length. One encoding involves a description of each point separately; other, perhaps shorter, encodings might involve a description of clusters of points together with how each point in a cluster can be described given the cluster it belongs to. The specific techniques described in this chapter do not explicitly make use of MDL principles, but the MDL method has

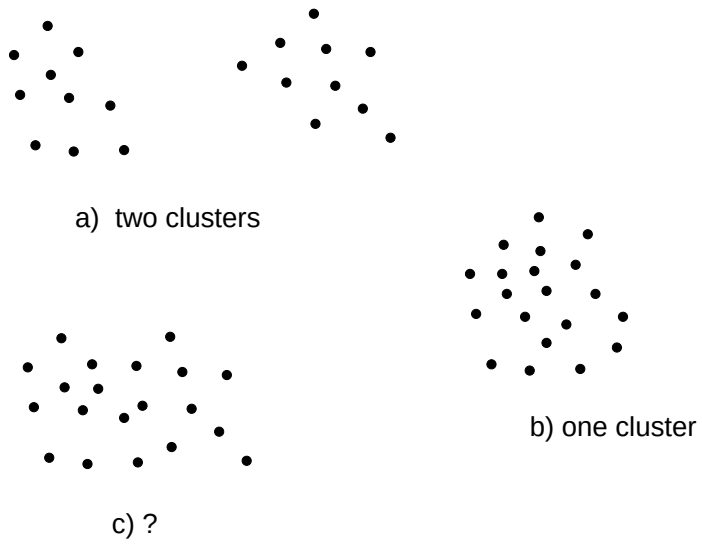


Figure 9.1: Unlabeled Patterns

been applied with success. One of the MDL-based methods, Autoclass II [Cheeseman, *et al.*, 1988] discovered a new classification of stars based on the properties of infrared sources.

Another type of unsupervised learning involves finding hierarchies of partitionings or clusters of clusters. A *hierarchical partition* is one in which Ξ is divided into mutually exclusive and exhaustive subsets, $\Xi_1, \ldots, \Xi_R$; each set, Ξ_i, $(i = 1, \ldots, R)$ is divided into mutually exclusive and exhaustive subsets, and so on. We show an example of such a hierarchical partition in Fig. 9.2. The hierarchical form is best displayed as a tree, as shown in Fig. 9.3. The tip nodes of the tree can further be expanded into their individual pattern elements. One application of such hierarchical partitions is in organizing individuals into taxonomic hierarchies such as those used in botany and zoology.

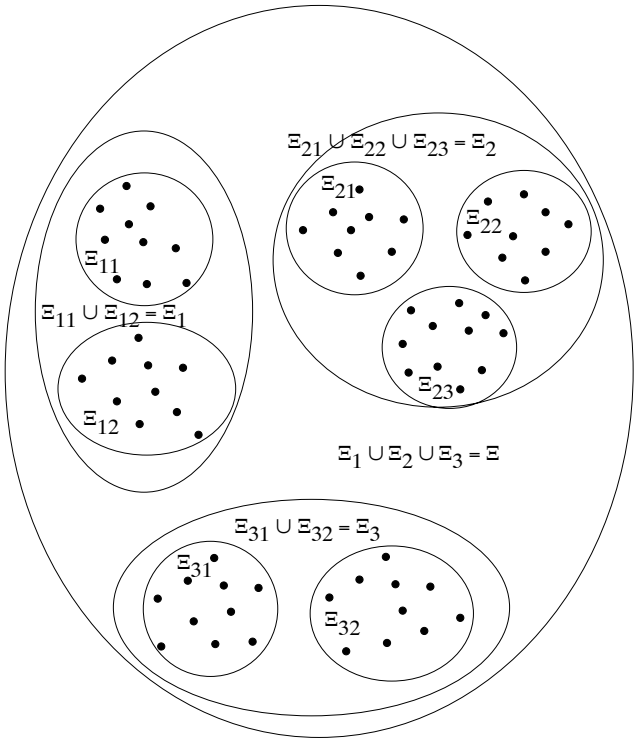


Figure 9.2: A Hierarchy of Clusters

9.2 Clustering Methods

9.2.1 A Method Based on Euclidean Distance

Most of the unsupervised learning methods use a measure of similarity between patterns in order to group them into clusters. The simplest of these involves defining a *distance* between patterns. For patterns whose features are numeric, the distance measure can be ordinary Euclidean distance between two points in an n-dimensional space.

There is a simple, iterative clustering method based on distance. It can be described as follows. Suppose we have R randomly chosen *cluster seekers*, $\mathbf{C}_1, \ldots, \mathbf{C}_R$. These are points in an n-dimensional space that we want to adjust so that they each move toward the center of one of the clusters of patterns. We present the (unlabeled) patterns in the training

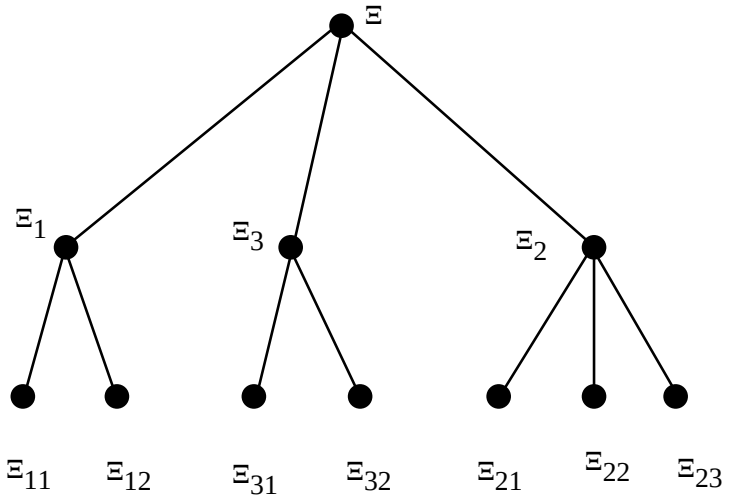


Figure 9.3: Displaying a Hierarchy as a Tree

set, Ξ, to the algorithm one-by-one. For each pattern, $\mathbf{X}_i$, presented, we find that cluster seeker, $\mathbf{C}_j$, that is closest to $\mathbf{X}_i$ and move it closer to $\mathbf{X}_i$:

$$\mathbf{C}_j \longleftarrow (1-\alpha_j)\mathbf{C}_j + \alpha_j \mathbf{X}_i$$

where α_j is a learning rate parameter for the j-th cluster seeker; it determines how far $\mathbf{C}_j$ is moved toward $\mathbf{X}_i$.

Refinements on this procedure make the cluster seekers move less far as training proceeds. Suppose each cluster seeker, $\mathbf{C}_j$, has a *mass*, m_j, equal to the number of times that it has moved. As a cluster seeker's mass increases it moves less far towards a pattern. For example, we might set $\alpha_j = 1/(1+m_j)$ and use the above rule together with $m_j \longleftarrow m_j + 1$. With this adjustment rule, a cluster seeker is always at the center of gravity (sample mean) of the set of patterns toward which it has so far moved. Intuitively, if a cluster seeker ever gets within some reasonably well clustered set of patterns (and if that cluster seeker is the only one so located), it will converge to the center of gravity of that cluster.

Once the cluster seekers have converged, the classifier implied by the now-labeled patterns in Ξ can be based on a Voronoi partitioning of the

space (based on distances to the various cluster seekers). This kind of classification, an example of which is shown in Fig. 9.4, can be implemented by a linear machine.

Georgy Fedoseevich Voronoi, was a Russian mathematician who lived from 1868 to 1909.

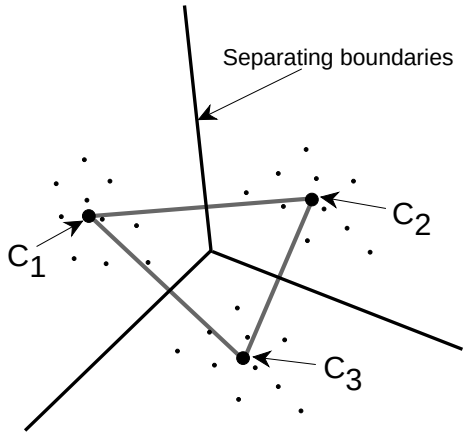


Figure 9.4: Minimum-Distance Classification

When basing partitioning on distance, we seek clusters whose patterns are as close together as possible. We can measure the *badness*, V, of a cluster of patterns, $\{\mathbf{X}_i\}$, by computing its *sample variance* defined by:

$$V = (1/K) \sum_i (\mathbf{X}_i - \mathbf{M})^2$$

where $\mathbf{M}$ is the sample mean of the cluster, which is defined to be:

$$\mathbf{M} = (1/K) \sum_i \mathbf{X}_i$$

and K is the number of points in the cluster.

We would like to partition a set of patterns into clusters such that the sum of the sample variances (badnesses) of these clusters is small. Of course if we have one cluster for each pattern, the sample variances will all be zero, so we must arrange that our measure of the badness of a partition must increase with the number of clusters. In this way, we can seek a trade-off between the variances of the clusters and the number of them in a way somewhat similar to the principle of minimal description length discussed earlier.

Elaborations of our basic cluster-seeking procedure allow the number of cluster seekers to vary depending on the distances between them and depending on the sample variances of the clusters. For example, if the distance, d_{ij}, between two cluster seekers, $\mathbf{C}_i$ and $\mathbf{C}_j$, ever falls below some threshold ε, then we can replace them both by a single cluster seeker placed at their center of gravity (taking into account their respective masses). In this way we can decrease the overall badness of a partition by reducing the number of clusters for comparatively little penalty in increased variance.

On the other hand, if any of the cluster seekers, say $\mathbf{C}_i$, defines a cluster whose sample variance is larger than some amount δ, then we can place a new cluster seeker, $\mathbf{C}_j$, at some random location somewhat adjacent to $\mathbf{C}_i$ and reset the masses of both $\mathbf{C}_i$ and $\mathbf{C}_j$ to zero. In this way the badness of the partition might ultimately decrease by decreasing the total sample variance with comparatively little penalty for the additional cluster seeker. The values of the parameters ε and δ are set depending on the relative weights given to sample variances and numbers of clusters.

In distance-based methods, it is important to scale the components of the pattern vectors. The variation of values along some dimensions of the pattern vector may be much different than that of other dimensions. One commonly used technique is to compute the standard deviation (*i.e.*, the square root of the variance) of each of the components over the entire training set and normalize the values of the components so that their adjusted standard deviations are equal.

9.2.2 A Method Based on Probabilities

Suppose we have a partition of the training set, Ξ, into R mutually exclusive and exhaustive clusters, $C_1, \ldots, C_R$. We can decide to which of these clusters some arbitrary pattern, $\mathbf{X}$, should be assigned by selecting the C_i for which the probability, $p(C_i|\mathbf{X})$, is largest, providing $p(C_i|\mathbf{X})$ is larger than some fixed threshold, δ. As we saw earlier, we can use Bayes rule and base our decision on maximizing $p(\mathbf{X}|C_i)p(C_i)$. Assuming conditional independence of the pattern components, x_i, the quantity to be maximized is:

$$S(\mathbf{X}, C_i) = p(x_1|C_i)p(x_2|C_i)\cdots p(x_n|C_i)p(C_i)$$

The $p(x_j|C_i)$ can be estimated from the sample statistics of the patterns in the clusters and then used in the above expression. (Recall the linear form that this formula took in the case of binary-valued components.)

We call $S(\mathbf{X}, C_i)$ the *similarity* of $\mathbf{X}$ to a cluster, C_i, of patterns. Thus, we assign $\mathbf{X}$ to the cluster to which it is most similar, providing the similarity is larger than δ.

Just as before, we can define the sample mean of a cluster, C_i, to be:

$$\mathbf{M}_i = (1/K_i) \sum_{\mathbf{X}_j \epsilon\ C_i} \mathbf{X}_j$$

where K_i is the number of patterns in C_i.

We can base an iterative clustering algorithm on this measure of similarity [Mahadevan & Connell, 1992]. It can be described as follows:

a. Begin with a set of unlabeled patterns Ξ and an empty list, L, of clusters.

b. For the next pattern, $\mathbf{X}$, in Ξ, compute $S(\mathbf{X}, C_i)$ for each cluster, C_i. (Initially, these similarities are all zero.) Suppose the largest of these similarities is $S(\mathbf{X}, C_{max})$.

   (a) If $S(\mathbf{X}, C_{max}) > \delta$, assign $\mathbf{X}$ to C_{max}. That is,

   $$C_{max} \longleftarrow C_{max} \cup \{\mathbf{X}\}$$

   Update the sample statistics $p(x_1|C_{max}), p(x_2|C_{max}), \ldots, p(x_n|C_{max})$, and $p(C_{max})$ to take the new pattern into account. Go to 3.

   (b) If $S(\mathbf{X}, C_{max}) \leq \delta$, create a new cluster, $C_{new} = \{\mathbf{X}\}$ and add C_{new} to L. Go to 3.

c. Merge any existing clusters, C_i and C_j if $(\mathbf{M}_i - \mathbf{M}_j)^2 < \varepsilon$. Compute new sample statistics $p(x_1|C_{merge}), p(x_2|C_{merge}), \ldots, p(x_n|C_{merge})$, and $p(C_{merge})$ for the merged cluster, $C_{merge} = C_i \cup C_j$.

d. If the sample statistics of the clusters have not changed during an entire iteration through Ξ, then terminate with the clusters in L; otherwise go to 2.

The value of the parameter δ controls the number of clusters. If δ is high, there will be a large number of clusters with few patterns in each cluster. For small values of δ, there will be a small number of clusters with many patterns in each cluster. Similarly, the larger the value of ε, the smaller the number clusters that will be found.

Designing a classifier based on the patterns labeled by the partitioning is straightforward. We assign any pattern, $\mathbf{X}$, to that category that maximizes $S(\mathbf{X}, C_i)$.

Mention "k-means and "EM" methods.

9.3 Hierarchical Clustering Methods

9.3.1 A Method Based on Euclidean Distance

Suppose we have a set, Ξ, of unlabeled training patterns. We can form a hierarchical classification of the patterns in Ξ by a simple *agglomerative* method. (The description of this algorithm is based on an unpublished manuscript by Pat Langley.) Our description here gives the general idea; we leave it to the reader to generate a precise algorithm.

We first compute the Euclidean distance between all pairs of patterns in Ξ. (Again, appropriate scaling of the dimensions is assumed.) Suppose the smallest distance is between patterns $\mathbf{X}_i$ and $\mathbf{X}_j$. We collect $\mathbf{X}_i$ and $\mathbf{X}_j$ into a cluster, C, eliminate $\mathbf{X}_i$ and $\mathbf{X}_j$ from Ξ and replace them by a *cluster vector*, $\mathbf{C}$, equal to the average of $\mathbf{X}_i$ and $\mathbf{X}_j$. Next we compute the Euclidean distance again between all pairs of points in Ξ. If the smallest distance is between pairs of patterns, we form a new cluster, C, as before and replace the pair of patterns in Ξ by their average. If the shortest distance is between a pattern, $\mathbf{X}_i$, and a cluster vector, $\mathbf{C}_j$ (representing a cluster, C_j), we form a new cluster, C, consisting of the union of C_j and $\{\mathbf{X}_i\}$. In this case, we replace $\mathbf{C}_j$ and $\mathbf{X}_i$ in Ξ by their (appropriately weighted) average and continue. If the shortest distance is between two cluster vectors, $\mathbf{C}_i$ and $\mathbf{C}_j$, we form a new cluster, C, consisting of the union of C_i and C_j. In this case, we replace $\mathbf{C}_i$ and $\mathbf{C}_j$ by their (appropriately weighted) average and continue. Since we reduce the number of points in Ξ by one each time, we ultimately terminate with a tree of clusters rooted in the cluster containing all of the points in the original training set.

An example of how this method aggregates a set of two dimensional patterns is shown in Fig. 9.5. The numbers associated with each cluster indicate the order in which they were formed. These clusters can be organized hierarchically in a binary tree with cluster 9 as root, clusters 7 and 8 as the two descendants of the root, and so on. A ternary tree could be formed instead if one searches for the three points in Ξ whose triangle defined by those patterns has minimal area.

9.3.2 A Method Based on Probabilities

A probabilistic quality measure for partitions

We can develop a measure of the goodness of a partitioning based on how accurately we can guess a pattern given only what partition it is in. Suppose we are given a partitioning of Ξ into R classes, $C_1, \ldots, C_R$. As before, we

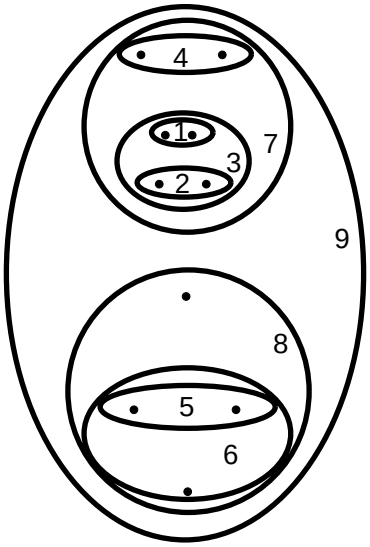


Figure 9.5: Agglommerative Clustering

can compute the sample statistics $p(x_i|C_k)$ which give probability values for each component given the class assigned to it by the partitioning. Suppose each component x_i of $\mathbf{X}$ can take on the values v_{ij}, where the index j steps over the domain of that component. We use the notation $p_i(v_{ij}|C_k) =$ probability$(x_i = v_{ij}|C_k)$.

Suppose we use the following probabilistic guessing rule about the values of the components of a vector $\mathbf{X}$ given only that it is in class k. Guess that $x_i = v_{ij}$ with probability $p_i(v_{ij}|C_k)$. Then, the probability that we guess the i-th component correctly is:

$$\sum_j \text{probability(guess is } v_{ij}) p_i(v_{ij}|C_k) = \sum_j \left[p_i(v_{ij}|C_k)\right]^2$$

The average number of (the n) components whose values are guessed correctly by this method is then given by the sum of these probabilities over all of the components of $\mathbf{X}$:

$$\sum_i \sum_j \left[p_i(v_{ij}|C_k)\right]^2$$

Given our partitioning into R classes, the goodness measure, G, of this partitioning is the average of the above expression over all classes:

$$G = \sum_k p(C_k) \sum_i \sum_j [p_i(v_{ij}|C_k)]^2$$

where $p(C_k)$ is the probability that a pattern is in class C_k. In order to penalize this measure for having a large number of classes, we divide it by R to get an overall "quality" measure of a partitioning:

$$Z = (1/R) \sum_k p(C_k) \sum_i \sum_j [p_i(v_{ij}|C_k)]^2$$

We give an example of the use of this measure for a trivially simple clustering of the four three-dimensional patterns shown in Fig. 9.6. There are several different partitionings. Let's evaluate Z values for the following ones: $P_1 = \{a, b, c, d\}$, $P_2 = \{\{a, b\}, \{c, d\}\}$, $P_3 = \{\{a, c\}, \{b, d\}\}$, and $P_4 = \{\{a\}, \{b\}, \{c\}, \{d\}\}$. The first, P_1, puts all of the patterns into a single cluster. The sample probabilities $p_i(v_{i1} = 1)$ and $p_i(v_{i0} = 0)$ are all equal to $1/2$ for each of the three components. Summing over the values of the components (0 and 1) gives $(1/2)^2 + (1/2)^2 = 1/2$. Summing over the three components gives $3/2$. Averaging over all of the clusters (there is just one) also gives $3/2$. Finally, dividing by the number of clusters produces the final Z value of this partition, $Z(P_1) = 3/2$.

The second partition, P_2, gives the following sample probabilities:

$$p_1(v_{11} = 1|C_1) = 1$$

$$p_2(v_{21} = 1|C_1) = 1/2$$

$$p_3(v_{31} = 1|C_1) = 1$$

Summing over the values of the components (0 and 1) gives $(1)^2 + (0)^2 = 1$ for component 1, $(1/2)^2 + (1/2)^2 = 1/2$ for component 2, and $(1)^2 + (0)^2 = 1$ for component 3. Summing over the three components gives 2 1/2 for class 1. A similar calculation also gives 2 1/2 for class 2. Averaging over the two clusters also gives 2 1/2. Finally, dividing by the number of clusters produces the final Z value of this partition, $Z(P_2) = 1\ 1/4$, not quite as high as $Z(P_1)$.

Similar calculations yield $Z(P_3) = 1$ and $Z(P_4) = 3/4$, so this method of evaluating partitions would favor placing all patterns in a single cluster.

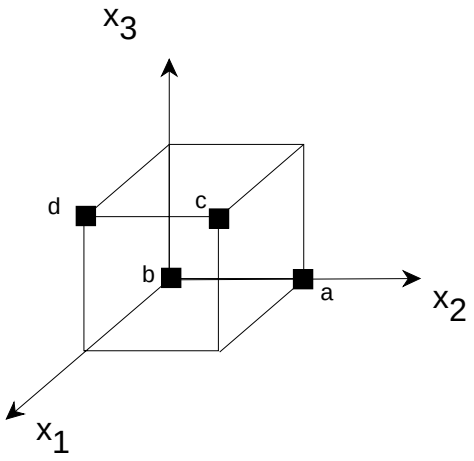


Figure 9.6: Patterns in 3-Dimensional Space

An iterative method for hierarchical clustering

Evaluating all partitionings of m patterns and then selecting the best would be computationally intractable. The following iterative method is based on a hierarchical clustering procedure called COBWEB [Fisher, 1987]. The procedure grows a tree each node of which is labeled by a set of patterns. At the end of the process, the root node contains all of the patterns in Ξ. The successors of the root node will contain mutually exclusive and exhaustive subsets of Ξ. In general, the successors of a node, η, are labeled by mutually exclusive and exhaustive subsets of the pattern set labelling node η. The tips of the tree will contain singleton sets. The method uses Z values to place patterns at the various nodes; sample statistics are used to update the Z values whenever a pattern is placed at a node. The algorithm is as follows:

a. We start with a tree whose root node contains all of the patterns in Ξ and a single empty successor node. We arrange that at all times during the process every non-empty node in the tree has (besides any other successors) exactly one empty successor.

b. Select a pattern $\mathbf{X}_i$ in Ξ (if there are no more patterns to select, terminate).

c. Set μ to the root node.

d. For each of the successors of μ (including the empty successor!), calculate the *best host* for $\mathbf{X}_i$. A best host is determined by tentatively placing $\mathbf{X}_i$ in one of the successors and calculating the resulting Z value for each one of these ways of accomodating $\mathbf{X}_i$. The best host corresponds to the assignment with the highest Z value.

e. If the best host is an empty node, η, we place $\mathbf{X}_i$ in η, generate an empty successor node of η, generate an empty sibling node of η, and go to 2.

f. If the best host is a non-empty, singleton (tip) node, η, we place $\mathbf{X}_i$ in η, create one successor node of η containing the singleton pattern that was in η, create another successor node of η containing $\mathbf{X}_i$, create an empty successor node of η, create empty successor nodes of the new non-empty successors of η, and go to 2.

g. If the best host is a non-empty, non-singleton node, η, we place $\mathbf{X}_i$ in η, set μ to η, and go to 4.

This process is rather sensitive to the order in which patterns are presented. To make the final classification tree less order dependent, the COBWEB procedure incorporates node *merging* and *splitting*.

Node merging:

It may happen that two nodes having the same parent could be merged with an overall increase in the quality of the resulting classification performed by the successors of that parent. Rather than try all pairs to merge, a good heuristic is to attempt to merge the two best hosts. When such a merging improves the Z value, a new node containing the union of the patterns in the merged nodes replaces the merged nodes, and the two nodes that were merged are installed as successors of the new node.

Node splitting:

A heuristic for node splitting is to consider replacing the best host among a group of siblings by that host's successors. This operation is performed only if it increases the Z value of the classification performed by a group of siblings.

Example results from COBWEB

We mention two experiments with COBWEB. In the first, the program attempted to find two categories (we will call them *Class 1* and *Class 2*) of

United States Senators based on their votes (*yes* or *no*) on six issues. After the clusters were established, the majority vote in each class was computed. These are shown in the table below.

Issue	Class 1	Class 2
Toxic Waste	yes	no
Budget Cuts	yes	no
SDI Reduction	no	yes
Contra Aid	yes	no
Line-Item Veto	yes	no
MX Production	yes	no

In the second experiment, the program attempted to classify soybean diseases based on various characteristics. COBWEB grouped the diseases in the taxonomy shown in Fig. 9.7.

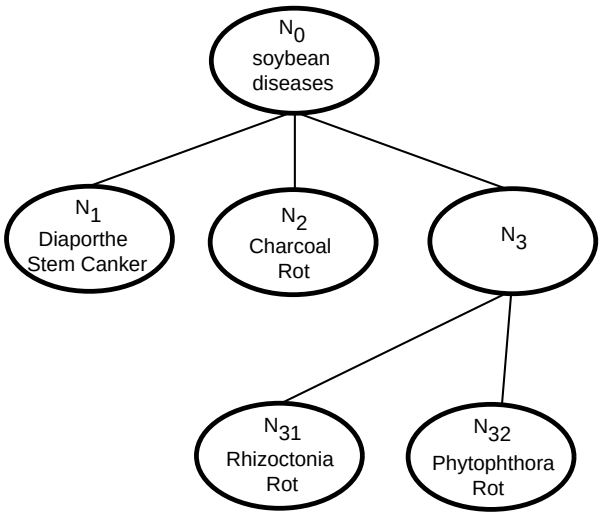


Figure 9.7: Taxonomy Induced for Soybean Diseases

9.4 Bibliographical and Historical Remarks

To be added.

Chapter 10

Temporal-Difference Learning

10.1 Temporal Patterns and Prediction Problems

In this chapter, we consider problems in which we wish to learn to predict the future value of some quantity, say z, from an n-dimensional input pattern, $\mathbf{X}$. In many of these problems, the patterns occur in temporal sequence, $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_i, \mathbf{X}_{i+1}, \ldots, \mathbf{X}_m$, and are generated by a dynamical process. The components of $\mathbf{X}_i$ are features whose values are available at time, $t = i$. We distinguish two kinds of prediction problems. In one, we desire to predict the value of z at time $t = i + 1$ based on input $\mathbf{X}_i$ for every i. For example, we might wish to predict some aspects of tomorrow's weather based on a set of measurements made today. In the other kind of prediction problem, we desire to make a sequence of predictions about the value of z at some *fixed* time, say $t = m + 1$, based on each of the $\mathbf{X}_i$, $i = 1, \ldots, m$. For example, we might wish to make a series of predictions about some aspect of the weather on next New Year's Day, based on measurements taken every day before New Year's. Sutton [Sutton, 1988] has called this latter problem, *multi-step prediction*, and that is the problem we consider here. In multi-step prediction, we might expect that the prediction accuracy should get better and better as i increases toward m.

10.2 Supervised and Temporal-Difference Methods

A training method that naturally suggests itself is to use the actual value of z at time $m+1$ (once it is known) in a supervised learning procedure using a sequence of training patterns, $\{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_i, \mathbf{X}_{i+1}, \ldots, \mathbf{X}_m\}$. That is, we seek to learn a function, f, such that $f(\mathbf{X}_i)$ is as close as possible to z for each i. Typically, we would need a training set, Ξ, consisting of several such sequences. We will show that a method that is better than supervised learning for some important problems is to base learning on the difference between $f(\mathbf{X}_{i+1})$ and $f(\mathbf{X}_i)$ rather than on the difference between z and $f(\mathbf{X}_i)$. Such methods involve what is called *temporal-difference (TD) learning.*

We assume that our prediction, $f(\mathbf{X})$, depends on a vector of modifiable weights, $\mathbf{W}$. To make that dependence explicit, we write $f(\mathbf{X}, \mathbf{W})$. For supervised learning, we consider procedures of the following type: For each $\mathbf{X}_i$, the prediction $f(\mathbf{X}_i, \mathbf{W})$ is computed and compared to z, and the learning rule (whatever it is) computes the change, $(\Delta \mathbf{W}_i)$, to be made to $\mathbf{W}$. Then, taking into account the weight changes for each pattern in a sequence all at once after having made all of the predictions with the old weight vector, we change $\mathbf{W}$ as follows:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} (\Delta \mathbf{W})_i$$

Whenever we are attempting to minimize the squared error between z and $f(\mathbf{X}_i, \mathbf{W})$ by gradient descent, the weight-changing rule for each pattern is:

$$(\Delta \mathbf{W})_i = c(z - f_i) \frac{\partial f_i}{\partial \mathbf{W}}$$

where c is a learning rate parameter, f_i is our prediction of z, $f(\mathbf{X}_i, \mathbf{W})$, at time $t = i$, and $\frac{\partial f_i}{\partial \mathbf{W}}$ is, by definition, the vector of partial derivatives $(\frac{\partial f_i}{\partial w_1}, \ldots, \frac{\partial f_i}{\partial w_i}, \ldots, \frac{\partial f_i}{\partial w_n})$ in which the w_i are the individual components of $\mathbf{W}$. (The expression $\frac{\partial f_i}{\partial \mathbf{W}}$ is sometimes written $\nabla_{\mathbf{W}} f_i$.) The reader will recall that we used an equivalent expression for $(\Delta \mathbf{W})_i$ in deriving the backpropagation formulas used in training multi-layer neural networks.

The Widrow-Hoff rule results when $f(\mathbf{X}, \mathbf{W}) = \mathbf{X} \bullet \mathbf{W}$. Then:

$$(\Delta \mathbf{W})_i = c(z - f_i)\mathbf{X}_i$$

An interesting form for $(\Delta \mathbf{W})_i$ can be developed if we note that

$$(z - f_i) = \sum_{k=i}^{m} (f_{k+1} - f_k)$$

where we define $f_{m+1} = z$. Substituting in our formula for $(\Delta \mathbf{W})_i$ yields:

$$(\Delta \mathbf{W})_i = c(z - f_i) \frac{\partial f_i}{\partial \mathbf{W}}$$

$$= c \frac{\partial f_i}{\partial \mathbf{W}} \sum_{k=i}^{m} (f_{k+1} - f_k)$$

In this form, instead of using the difference between a prediction and the value of z, we use the differences between successive predictions—thus the phrase *temporal-difference (TD)* learning.

In the case when $f(\mathbf{X}, \mathbf{W}) = \mathbf{X} \bullet \mathbf{W}$, the temporal difference form of the Widrow-Hoff rule is:

$$(\Delta \mathbf{W})_i = c\mathbf{X}_i \sum_{k=i}^{m} (f_{k+1} - f_k)$$

One reason for writing $(\Delta \mathbf{W})_i$ in temporal-difference form is to permit an interesting generalization as follows:

$$(\Delta \mathbf{W})_i = c \frac{\partial f_i}{\partial \mathbf{W}} \sum_{k=i}^{m} \lambda^{(k-i)} (f_{k+1} - f_k)$$

where $0 < \lambda \leq 1$. Here, the λ term gives exponentially decreasing weight to differences later in time than $t = i$. When $\lambda = 1$, we have the same rule with which we began—weighting all differences equally, but as $\lambda \to 0$, we weight only the $(f_{i+1} - f_i)$ difference. With the λ term, the method is called TD(λ).

It is interesting to compare the two extreme cases:
For TD(0):

$$(\Delta \mathbf{W})_i = c(f_{i+1} - f_i) \frac{\partial f_i}{\partial \mathbf{W}}$$

For TD(1):

$$(\Delta \mathbf{W})_i = c(z - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

Both extremes can be handled by the same learning mechanism; only the error term is different. In TD(0), the error is the difference between successive predictions, and in TD(1), the error is the difference between the finally revealed value of z and the prediction. Intermediate values of λ take into account differently weighted differences between future pairs of successive predictions.

Only TD(1) can be considered a pure *supervised* learning procedure, sensitive to the final value of z provided by the teacher. For $\lambda < 1$, we have various degrees of unsupervised learning, in which the prediction function strives to make each prediction more like successive ones (whatever they might be). We shall soon see that these unsupervised procedures result in better learning than do the supervised ones for an important class of problems.

10.3 Incremental Computation of the $(\Delta \mathbf{W})_i$

We can rewrite our formula for $(\Delta \mathbf{W})_i$, namely

$$(\Delta \mathbf{W})_i = c\frac{\partial f_i}{\partial \mathbf{W}} \sum_{k=i}^{m} \lambda^{(k-i)}(f_{k+1} - f_k)$$

to allow a type of incremental computation. First we write the expression for the weight change rule that takes into account all of the $(\Delta \mathbf{W})_i$:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} c\frac{\partial f_i}{\partial \mathbf{W}} \sum_{k=i}^{m} \lambda^{(k-i)}(f_{k+1} - f_k)$$

Interchanging the order of the summations yields:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{k=1}^{m} c \sum_{i=1}^{k} \lambda^{(k-i)}(f_{k+1} - f_k)\frac{\partial f_i}{\partial \mathbf{W}}$$

$$= \mathbf{W} + \sum_{k=1}^{m} c(f_{k+1} - f_k) \sum_{i=1}^{k} \lambda^{(k-i)}\frac{\partial f_i}{\partial \mathbf{W}}$$

Interchanging the indices k and i finally yields:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

If, as earlier, we want to use an expression of the form $\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} (\Delta \mathbf{W})_i$, we see that we can write:

$$(\Delta \mathbf{W})_i = c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

Now, if we let $e_i = \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$, we can develop a computationally efficient recurrence equation for e_{i+1} as follows:

$$e_{i+1} = \sum_{k=1}^{i+1} \lambda^{(i+1-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

$$= \frac{\partial f_{i+1}}{\partial \mathbf{W}} + \sum_{k=1}^{i} \lambda^{(i+1-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

$$= \frac{\partial f_{i+1}}{\partial \mathbf{W}} + \lambda e_i$$

Rewriting $(\Delta \mathbf{W})_i$ in these terms, we obtain:

$$(\Delta \mathbf{W})_i = c(f_{i+1} - f_i) e_i$$

where:

$$e_1 = \frac{\partial f_1}{\partial \mathbf{W}}$$

$$e_2 = \frac{\partial f_2}{\partial \mathbf{W}} + \lambda e_1$$

etc.

Quoting Sutton [Sutton, 1988, page 15] (about a different equation, but the quote applies equally well to this one):

> "... this equation can be computed incrementally, because each $(\Delta \mathbf{W})_i$ depends only on a pair of successive predictions and on the [weighted] sum of all past values for $\frac{\partial f_i}{\partial \mathbf{W}}$. This saves substantially on memory, because it is no longer necessary to individually remember all past values of $\frac{\partial f_i}{\partial \mathbf{W}}$."

10.4 An Experiment with TD Methods

TD prediction methods [especially TD(0)] are well suited to situations in which the patterns are generated by a dynamic process. In that case, sequences of temporally presented patterns contain important information that is ignored by a conventional supervised method such as the Widrow-Hoff rule. Sutton [Sutton, 1988, page 19] gives an interesting example involving a random walk, which we repeat here. In Fig. 10.1, sequences of vectors, $\mathbf{X}$, are generated as follows: We start with vector $\mathbf{X}_D$; the next vector in the sequence is equally likely to be one of the adjacent vectors in the diagram. If the next vector is $\mathbf{X}_C$ (or $\mathbf{X}_E$), the next one after that is equally likely to be one of the vectors adjacent to $\mathbf{X}_C$ (or $\mathbf{X}_E$). When $\mathbf{X}_B$ is in the sequence, it is equally likely that the sequence terminates with $z = 0$ or that the next vector is $\mathbf{X}_C$. Similarly, when $\mathbf{X}_F$ is in the sequence, it is equally likely that the sequence terminates with $z = 1$ or that the next vector is $\mathbf{X}_E$. Thus the sequences are random, but they always start with $\mathbf{X}_D$. Some sample sequences are shown in the figure. This random walk is an example of a *Markov process*; transitions from state i to state j occur with probabilities that depend only on i and j.

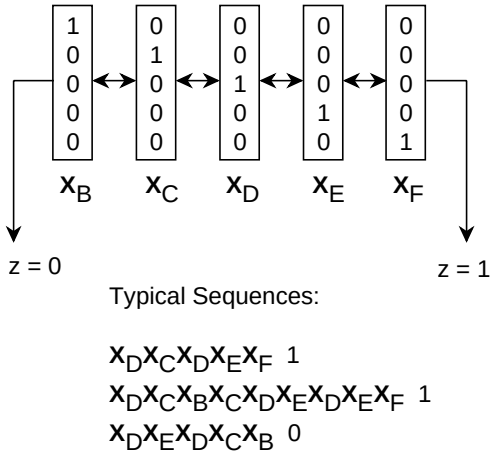


Figure 10.1: A Markov Process

Given a set of sequences generated by this process as a training set, we want to be able to predict the value of z for each $\mathbf{X}$ in a test sequence. We

assume that the learning system does not know the transition probabilities.

For his experiments with this process, Sutton used a linear predictor, that is $f(\mathbf{X}, \mathbf{W}) = \mathbf{X} \bullet \mathbf{W}$. The learning problem is to find a weight vector, $\mathbf{W}$, that minimizes the mean-squared error between z and the predicted value of z. Given the five different values that $\mathbf{X}$ can take on, we have the following predictions: $f(\mathbf{X}_B) = w_1$, $f(\mathbf{X}_C) = w_2$, $f(\mathbf{X}_D) = w_3$, $f(\mathbf{X}_E) = w_4$, $f(\mathbf{X}_F) = w_5$, where w_i is the i-th component of the weight vector. (Note that the values of the predictions are not limited to 1 or 0—even though z can only have one of those values—because we are minimizing mean-squared error.) After training, these predictions will be compared with the optimal ones—given the transition probabilities.

The experimental setup was as follows: ten random sequences were generated using the transition probabilities. Each of these sequences was presented in turn to a TD(λ) method for various values of λ. Weight vector increments, $(\Delta \mathbf{W})_i$, were computed after each pattern presentation but no weight changes were made until all ten sequences were presented. The weight vector increments were summed after all ten sequences were presented, and this sum was used to change the weight vector to be used for the next pass through the ten sequences. This process was repeated over and over (using the same training sequences) until (quoting Sutton) "the procedure no longer produced any significant changes in the weight vector. For small c, the weight vector always converged in this way, and always to the same final value [for 100 different training sets of ten random sequences], independent of its initial value." (Even though, for fixed, small c, the weight vector always converged to the same vector, it might converge to a somewhat different vector for different values of c.)

After convergence, the predictions made by the final weight vector are compared with the optimal predictions made using the transition probabilities. These optimal predictions are simply $p(z = 1|\mathbf{X})$. We can compute these probabilities to be 1/6, 1/3, 1/2, 2/3, and 5/6 for $\mathbf{X}_B$, $\mathbf{X}_C$, $\mathbf{X}_D$, $\mathbf{X}_E$, $\mathbf{X}_F$, respectively. The root-mean-squared differences between the best learned predictions (over all c) and these optimal ones are plotted in Fig. 10.2 for seven different values of λ. (For each data point, the standard error is approximately $\sigma = 0.01$.)

Notice that the Widrow-Hoff procedure does not perform as well as other versions of TD(λ) for $\lambda < 1$! Quoting [Sutton, 1988, page 21]:

> "This result contradicts conventional wisdom. It is well known that, under repeated presentations, the Widrow-Hoff procedure minimizes the RMS error between its predictions and the actual outcomes in the training set ([Widrow & Stearns, 1985]).

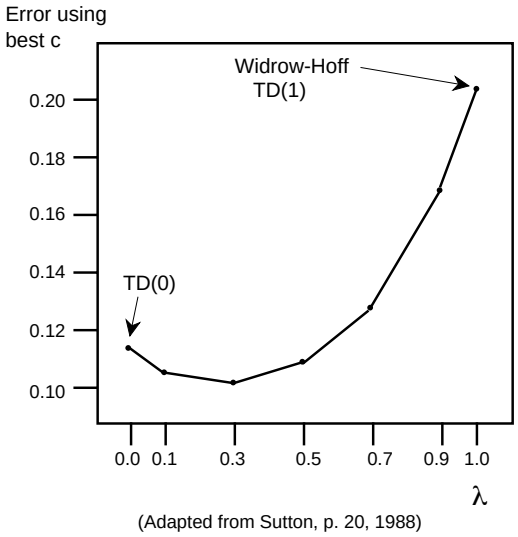


Figure 10.2: Prediction Errors for TD(λ)

> How can it be that this optimal method peformed worse than all the TD methods for $\lambda < 1$? The answer is that the Widrow-Hoff procedure only minimizes error *on the training set*; it does not necessarily minimize error for future experience. [Later] we prove that in fact it is linear TD(0) that converges to what can be considered the optimal estimates for matching future experience—those consistent with the maximum-likelihood estimate of the underlying Markov process."

10.5 Theoretical Results

It is possible to analyze the performance of the linear-prediction TD(λ) methods on Markov processes. We state some theorems here without proof.

Theorem 10.1 (Sutton, page 24, 1988) *For any absorbing Markov chain, and for any linearly independent set of observation vectors* $\{\mathbf{X}_i\}$ *for the non-terminal states, there exists an* $\varepsilon > 0$ *such that for all positive* $c < \varepsilon$ *and for any initial weight vector, the predictions of linear TD(0) (with*

weight updates after each sequence) converge in expected value to the optimal (maximum likelihood) predictions of the true process.

Even though the expected values of the predictions converge, the predictions themselves do not converge but vary around their expected values depending on their most recent experience. Sutton conjectures that if c is made to approach 0 as training progresses, the variance of the predictions will approach 0 also.

Dayan [Dayan, 1992] has extended the result of Theorem 9.1 to TD(λ) for arbitrary λ between 0 and 1. (Also see [Dayan & Sejnowski, 1994].)

10.6 Intra-Sequence Weight Updating

Our standard weight updating rule for TD(λ) methods is:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

where the weight update occurs *after* an entire sequence is observed. To make the method truly incremental (in analogy with weight updating rules for neural nets), it would be desirable to change the weight vector after every pattern presentation. The obvious extension is:

$$\mathbf{W}_{i+1} \longleftarrow \mathbf{W}_i + c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

where f_{i+1} is computed before making the weight change; that is, $f_{i+1} = f(\mathbf{X}_{i+1}, \mathbf{W}_i)$. But that would make $f_i = f(\mathbf{X}_i, \mathbf{W}_{i-1})$, and such a rule would make the prediction difference, namely $(f_{i+1} - f_i)$, sensitive both to changes in $\mathbf{X}$ and changes in $\mathbf{W}$ and could lead to instabilities. Instead, we modify the rule so that, for every pair of predictions, $f_{i+1} = f(\mathbf{X}_{i+1}, \mathbf{W}_i)$ and $f_i = f(\mathbf{X}_i, \mathbf{W}_i)$. This version of the rule has been used in practice with excellent results.

For TD(0) and linear predictors, the rule is:

$$\mathbf{W}_{i+1} = \mathbf{W}_i + c(f_{i+1} - f_i)\mathbf{X}_i$$

The rule is implemented as follows:

a. Initialize the weight vector, $\mathbf{W}$, arbitrarily.

b. For $i = 1, ..., m$, **do**:

   (a) $f_i \longleftarrow \mathbf{X}_i \bullet \mathbf{W}$
   (We compute f_i anew each time through rather than use the value of f_{i+1} the previous time through.)

   (b) $f_{i+1} \longleftarrow \mathbf{X}_{i+1} \bullet \mathbf{W}$

   (c) $d_{i+1} \longleftarrow f_{i+1} - f_i$

   (d) $\mathbf{W} \longleftarrow \mathbf{W} + c\ d_{i+1}\mathbf{X}_i$
   (If f_i were computed again with this changed weight vector, its value would be closer to f_{i+1} as desired.)

The linear TD(0) method can be regarded as a technique for training a very simple network consisting of a single dot product unit (and no threshold or sigmoid function). TD methods can also be used in combination with backpropagation to train neural networks. For TD(0) we change the network weights according to the expression:

$$\mathbf{W}_{i+1} = \mathbf{W}_i + c(f_{i+1} - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

The only change that must be made to the standard backpropagation weight-changing rule is that the difference term between the desired output and the output of the unit in the final (k-th) layer, namely $(d - f^{(k)})$, must be replaced by a difference term between successive outputs, $(f_{i+1} - f_i)$. This change has a direct effect only on the expression for $\delta^{(k)}$ which becomes:

$$\delta^{(k)} = 2(f'^{(k)} - f^{(k)})f^{(k)}(1 - f^{(k)})$$

where $f'^{(k)}$ and $f^{(k)}$ are two successive outputs of the network.

The weight changing rule for the i-th weight vector in the j-th layer of weights has the same form as before, namely:

$$\mathbf{W}_i^{(j)} \longleftarrow \mathbf{W}_i^{(j)} + c\delta_i^{(j)}\mathbf{X}^{(j-1)}$$

where the $\delta_i^{(j)}$ are given recursively by:

$$\delta_i^{(j)} = f_i^{(j)}(1 - f_i^{(j)}) \sum_{l=1}^{m_{j+1}} \delta_l^{(j+1)} w_{il}^{(j+1)}$$

and $w_{il}^{(j+1)}$ is the l-th component of the i-th weight vector in the $(j+1)$-th layer of weights. Of course, here also it is assumed that $f'^{(k)}$ and $f^{(k)}$ are computed using the same weights and *then* the weights are changed. In the next section we shall see an interesting example of this application of TD learning.

10.7 An Example Application: TD-gammon

A program called TD-gammon [Tesauro, 1992] learns to play backgammon by training a neural network via temporal-difference methods. The structure of the neural net, and its coding is as shown in Fig. 10.3. The network is trained to minimize the error between actual payoff and estimated payoff, where the actual payoff is defined to be $d_f = p_1 + 2p_2 - p_3 - 2p_4$, and the p_i are the actual probabilities of the various outcomes as defined in the figure.

TD-gammon learned by using the network to select that move that results in the best predicted payoff. That is, at any stage of the game some finite set of moves is possible and these lead to the set, $\{\mathbf{X}\}$, of new board positions. Each member of this set is evaluated by the network, and the one with the largest predicted payoff is selected if it is white's move (and the smallest if it is black's). The move is made, and the network weights are adjusted to make the predicted payoff from the original position closer to that of the resulting position.

The weight adjustment procedure combines temporal-difference (TD(λ)) learning with backpropagation. If d_t is the network's estimate of the payoff at time t (before a move is made), and d_{t+1} is the estimate at time $t+1$ (after a move is made), the weight adjustment rule is:

$$\Delta \mathbf{W}_t = c(d_{t+1} - d_t) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial d_k}{\partial \mathbf{W}}$$

where $\mathbf{W}_t$ is a vector of *all* weights in the network at time t, and $\frac{\partial d_k}{\partial \mathbf{W}}$ is the gradient of d_k in this weight space. (For a layered, feedforward network, such as that of TD-gammon, the weight changes for the weight vectors in each layer can be expressed in the usual manner.)

To make the special cases clear, recall that for TD(0), the network would be trained so that, for all t, its output, d_t, for input $\mathbf{X}_t$ tended toward its expected output, d_{t+1}, for input $\mathbf{X}_{t+1}$. For TD(1), the network would be trained so that, for all t, its output, d_t, for input $\mathbf{X}_t$ tended toward the expected final payoff, d_f, given that input. The latter case is the same as the Widrow-Hoff rule.

After about 200,000 games the following results were obtained. TD-gammon (with 40 hidden units, $\lambda = 0.7$, and $c = 0.1$) won 66.2% of 10,000 games against SUN Microsystems Gammontool and 55% of 10,000 games against a neural network trained using expert moves. Commenting on a later version of TD-gammon, incorporating special features as inputs, Tesauro said: "It appears to be the strongest program ever seen by this author."

10.8 Bibliographical and Historical Remarks

To be added.

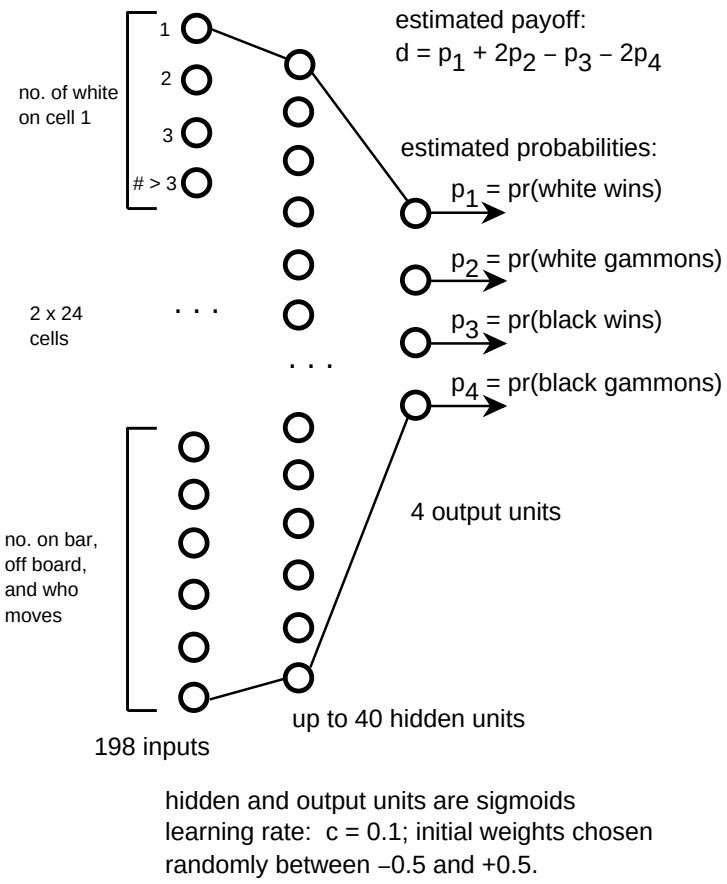

1
2
3
> 3
no. of white on cell 1
2 x 24 cells
no. on bar, off board, and who moves
198 inputs
up to 40 hidden units
4 output units
estimated payoff:
d = p1 + 2p2 – p3 – 2p4
estimated probabilities:
p1 = pr(white wins)
p2 = pr(white gammons)
p3 = pr(black wins)
p4 = pr(black gammons)
hidden and output units are sigmoids
learning rate: c = 0.1; initial weights chosen randomly between –0.5 and +0.5.

Figure 10.3: The TD-gammon Network

Chapter 11

Delayed-Reinforcement Learning

11.1 The General Problem

Imagine a robot that exists in an environment in which it can sense and act. Suppose (as an extreme case) that it has no idea about the effects of its actions. That is, it doesn't know how acting will change its sensory inputs. Along with its sensory inputs are "rewards," which it occasionally receives. How should it choose its actions so as to maximize its rewards over the long run? To maximize rewards, it will need to be able to predict how actions change inputs, and in particular, how actions lead to rewards.

We formalize the problem in the following way: The robot exists in an environment consisting of a set, $\mathcal{S}$, of states. We assume that the robot's sensory apparatus constructs an input vector, $\mathbf{X}$, from the environment, which informs the robot about which state the environment is in. For the moment, we will assume that the mapping from states to vectors is one-to-one, and, in fact, will use the notation $\mathbf{X}$ to refer to the state of the environment as well as to the input vector. When presented with an input vector, the robot decides which action from a set, $\mathcal{A}$, of actions to perform. Performing the action produces an effect on the environment—moving it to a new state. The new state results in the robot perceiving a new input vector, and the cycle repeats. We assume a discrete time model; the input vector at time $t = i$ is $\mathbf{X}_i$, the action taken at that time is a_i, and the expected reward, r_i, received at $t = i$ depends on the action taken and on the state, that is $r_i = r(\mathbf{X}_i, a_i)$. The learner's goal is to

find a *policy*, $\pi(\mathbf{X})$, that maps input vectors to actions in such a way that maximizes rewards accumulated over time. This type of learning is called *reinforcement learning*. The learner must find the policy by trial and error; it has no initial knowledge of the effects of its actions. The situation is as shown in Fig. 11.1.

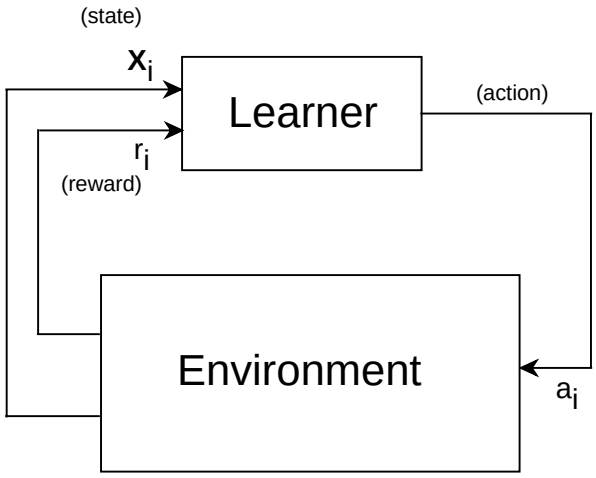


Figure 11.1: Reinforcement Learning

11.2 An Example

A "grid world," such as the one shown in Fig. 11.2 is often used to illustrate reinforcement learning. Imagine a robot initially in cell (2,3). The robot receives input vector (x_1, x_2) telling it what cell it is in; it is capable of four actions, n, e, s, w moving the robot one cell up, right, down, or left, respectively. It is rewarded one negative unit whenever it bumps into the wall or into the blocked cells. For example, if the input to the robot is (1,3), and the robot chooses action w, the next input to the robot is still (1,3) and it receives a reward of -1. If the robot lands in the cell marked G (for goal), it receives a reward of $+10$. Let's suppose that whenever the robot lands in the goal cell and gets its reward, it is immediately transported out to some random cell, and the quest for reward continues.

A *policy* for our robot is a specification of what action to take for every one of its inputs, that is, for every one of the cells in the grid. For example,

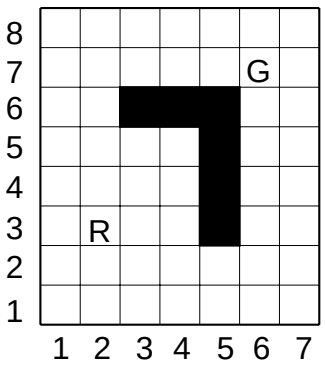


Figure 11.2: A Grid World

a component of such a policy would be "when in cell (3,1), move right." An *optimal policy* is a policy that maximizes long-term reward. One way of displaying a policy for our grid-world robot is by an arrow in each cell indicating the direction the robot should move when in that cell. In Fig. 11.3, we show an optimal policy displayed in this manner. In this chapter we will describe methods for learning optimal policies based on reward values received by the learner.

11.3 Temporal Discounting and Optimal Policies

In delayed reinforcement learning, one often assumes that rewards in the distant future are not as valuable as are more immediate rewards. This preference can be accomodated by a *temporal discount factor*, $0 \leq \gamma < 1$. The present value of a reward, r_i, occuring i time units in the future, is taken to be $\gamma^i r_i$. Suppose we have a policy $\pi(\mathbf{X})$ that maps input vectors into actions, and let $r_i^{\pi(\mathbf{X})}$ be the reward that will be received on the i-th time step after one begins executing policy π starting in state $\mathbf{X}$. Then the total reward accumulated over all time steps by policy π beginning in state $\mathbf{X}$ is:

$$V^{\pi}(\mathbf{X}) = \sum_{i=0}^{\infty} \gamma^i r_i^{\pi(\mathbf{X})}$$

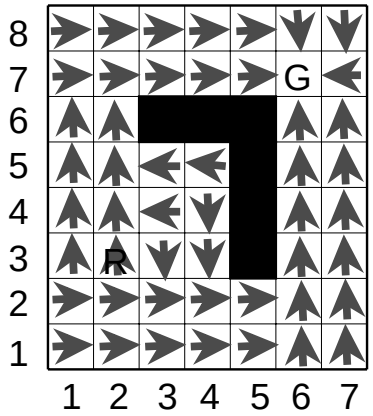


Figure 11.3: An Optimal Policy in the Grid World

One reason for using a temporal discount factor is so that the above sum will be finite. An optimal policy is one that maximizes $V^\pi(\mathbf{X})$ for all inputs, $\mathbf{X}$.

In general, we want to consider the case in which the rewards, r_i, are random variables and in which the effects of actions on environmental states are random. In Markovian environments, for example, the probability that action a in state $\mathbf{X}_i$ will lead to state $\mathbf{X}_j$ is given by a transition probability $p[\mathbf{X}_j|\mathbf{X}_i, a]$. Then, we will want to maximize *expected* future reward and would define $V^\pi(\mathbf{X})$ as:

$$V^\pi(\mathbf{X}) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i^{\pi(\mathbf{X})}\right]$$

In either case, we call $V^\pi(\mathbf{X})$ the *value* of policy π for input $\mathbf{X}$.

If the action prescribed by π taken in state $\mathbf{X}$ leads to state $\mathbf{X}'$ (randomly according to the transition probabilities), then we can write $V^\pi(\mathbf{X})$ in terms of $V^\pi(\mathbf{X}')$ as follows:

$$V^\pi(\mathbf{X}) = r[\mathbf{X}, \pi(\mathbf{X})] + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, \pi(\mathbf{X})] V^\pi(\mathbf{X}')$$

where (in summary):

$\gamma =$ the discount factor,

$V^{\pi}(\mathbf{X}) =$ the value of state $\mathbf{X}$ under policy π,

$r[\mathbf{X}, \pi(\mathbf{X})] =$ the expected immediate reward received when we execute the action prescribed by π in state $\mathbf{X}$, and

$p[\mathbf{X}'|\mathbf{X}, \pi(\mathbf{X})] =$ the probability that the environment transitions to state $\mathbf{X}'$ when we execute the action prescribed by π in state $\mathbf{X}$.

In other words, the value of state $\mathbf{X}$ under policy π is the expected value of the immediate reward received when executing the action recommended by π plus the average value (under π) of all of the states accessible from $\mathbf{X}$.

For an optimal policy, π^* (and no others!), we have the famous "optimality equation:"

$$V^{\pi^*}(\mathbf{X}) = \max_a \left[r(\mathbf{X}, a) + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, a] V^{\pi^*}(\mathbf{X}') \right]$$

The theory of dynamic programming (DP) [Bellman, 1957, Ross, 1983] assures us that there is at least one optimal policy, π^*, that satisfies this equation. DP also provides methods for calculating $V^{\pi^*}(\mathbf{X})$ and at least one π^*, assuming that we know the average rewards and the transition probabilities. If we knew the transition probabilities, the average rewards, and $V^{\pi^*}(\mathbf{X})$ for all $\mathbf{X}$ and a, then it would be easy to implement an optimal policy. We would simply select that a that maximizes $r(\mathbf{X}, a) + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, a] V^{\pi^*}(\mathbf{X}')$. That is,

$$\pi^*(\mathbf{X}) = \arg\max_a \left[r(\mathbf{X}, a) + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, a] V^{\pi^*}(\mathbf{X}') \right]$$

But, of course, we are assuming that we do not know these average rewards nor the transition probabilities, so we have to find a method that effectively learns them.

If we had a model of actions, that is, if we knew for every state, $\mathbf{X}$, and action a, which state, $\mathbf{X}'$ resulted, then we could use a method called *value iteration* to find an optimal policy. Value iteration works as follows: We begin by assigning, randomly, an *estimated value* $\hat{V}(\mathbf{X})$ to every state, $\mathbf{X}$. On the i-th step of the process, suppose we are at state $\mathbf{X}_i$ (that is, our input on the i-th step is $\mathbf{X}_i$), and that the estimated value of state $\mathbf{X}_i$ on the i-th step is $\hat{V}_i(\mathbf{X}_i)$. We then select that action a that maximizes the estimated value of the predicted subsequent state. Suppose this subsequent state

having the highest estimated value is $\mathbf{X}'_i$. Then we update the estimated value, $\hat{V}_i(\mathbf{X}_i)$, of state $\mathbf{X}_i$ as follows:

$$\hat{V}_i(\mathbf{X}) = (1 - c_i)\hat{V}_{i-1}(\mathbf{X}) + c_i\left[r_i + \gamma\hat{V}_{i-1}(\mathbf{X}'_i)\right]$$

if $\mathbf{X} = \mathbf{X}_i$,

$$= \hat{V}_{i-1}(\mathbf{X})$$

otherwise.

We see that this adjustment moves the value of $\hat{V}_i(\mathbf{X}_i)$ an increment (depending on c_i) closer to $\left[r_i + \gamma\hat{V}_i(\mathbf{X}'_i)\right]$. Assuming that $\hat{V}_i(\mathbf{X}'_i)$ is a good estimate for $V_i(\mathbf{X}'_i)$, then this adjustment helps to make the two estimates more consistent. Providing that $0 < c_i < 1$ and that we visit each state infinitely often, this process of value iteration will converge to the optimal values.

Discuss synchronous dynamic programming, asynchronous dynamic programming, and policy iteration.

11.4 Q-Learning

Watkins [Watkins, 1989] has proposed a technique that he calls *incremental dynamic programming*. Let $a;\pi$ stand for the policy that chooses action a once, and thereafter chooses actions according to policy π. We define:

$$Q^\pi(\mathbf{X}, a) = V^{a;\pi}(\mathbf{X})$$

Then the optimal value from state $\mathbf{X}$ is given by:

$$V^{\pi^*}(\mathbf{X}) = \max_a Q^{\pi^*}(\mathbf{X}, a)$$

This equation holds only for an optimal policy, π^*. The optimal policy is given by:

$$\pi^*(\mathbf{X}) = \arg\max_a Q^{\pi^*}(\mathbf{X}, a)$$

Note that if an action a makes $Q^\pi(\mathbf{X}, a)$ larger than $V^\pi(\mathbf{X})$, then we can improve π by changing it so that $\pi(\mathbf{X}) = a$. Making such a change is the basis for a powerful learning rule that we shall describe shortly.

Suppose action a in state $\mathbf{X}$ leads to state $\mathbf{X}'$. Then using the definitions of Q and V, it is easy to show that:

$$Q^{\pi}(\mathbf{X}, a) = r(\mathbf{X}, a) + \gamma E[V^{\pi}(\mathbf{X}')]$$

where $r(\mathbf{X}, a)$ is the average value of the immediate reward received when we execute action a in state $\mathbf{X}$. For an optimal policy (and no others), we have another version of the optimality equation in terms of Q values:

$$Q^{\pi^*}(\mathbf{X}, a) = \max_a \Big[r(\mathbf{X}, a) + \gamma E\Big[Q^{\pi^*}(\mathbf{X}', a) \Big] \Big]$$

for all actions, a, and states, $\mathbf{X}$. Now, if we had the optimal Q values (for all a and $\mathbf{X}$), then we could implement an optimal policy simply by selecting that action that maximized $r(\mathbf{X}, a) + \gamma E\big[Q^{\pi^*}(\mathbf{X}', a)\big]$.
That is,

$$\pi^*(\mathbf{X}) = \arg\max_a \Big[r(\mathbf{X}, a) + \gamma E\Big[Q^{\pi^*}(\mathbf{X}', a) \Big] \Big]$$

Watkins' proposal amounts to a TD(0) method of learning the Q values. We quote (with minor notational changes) from [Watkins & Dayan, 1992, page 281]:

> "In Q-Learning, the agent's experience consists of a sequence of distinct stages or *episodes*. In the i-th episode, the agent:
>
> - observes its current state $\mathbf{X}_i$,
> - selects [using the method described below] and performs an action a_i,
> - observes the subsequent state $\mathbf{X}'_i$,
> - receives an immediate reward r_i, and
> - adjusts its Q_{i-1} values using a learning factor c_i, according to:
>
> $$Q_i(\mathbf{X}, a) = (1 - c_i) Q_{i-1}(\mathbf{X}, a) + c_i [r_i + \gamma V_{i-1}(\mathbf{X}'_i)]$$
>
> if $\mathbf{X} = \mathbf{X}_i$ and $a = a_i$,
>
> $$= Q_{i-1}(\mathbf{X}, a)$$
>
> otherwise,

where

$$V_{i-1}(\mathbf{X}') = \max_b [Q_{i-1}(\mathbf{X}', b)]$$

is the best the agent thinks it can do from state $\mathbf{X}'$. ... The initial Q values, $Q_0(\mathbf{X}, a)$, for all states and actions are assumed given."

Using the current Q values, $Q_i(\mathbf{X}, a)$, the agent always selects that action that maximizes $Q_i(\mathbf{X}, a)$. Note that only the Q value corresponding to the state just exited and the action just taken is adjusted. And that Q value is adjusted so that it is closer (by an amount determined by c_i) to the sum of the immediate reward plus the discounted maximum (over all actions) of the Q values of the state just entered. If we imagine the Q values to be predictions of ultimate (infinite horizon) total reward, then the learning procedure described above is exactly a TD(0) method of learning how to predict these Q values. Q learning strengthens the usual TD methods, however, because TD (applied to reinforcement problems using value iteration) requires a one-step lookahead, using a model of the effects of actions, whereas Q learning does not.

A convenient notation (proposed by [Schwartz, 1993]) for representing the change in Q value is:

$$Q(\mathbf{X}, a) \stackrel{\beta}{\longleftarrow} r + \gamma V(\mathbf{X}')$$

where $Q(\mathbf{X}, a)$ is the new Q value for input $\mathbf{X}$ and action a, r is the immediate reward when action a is taken in response to input $\mathbf{X}$, $V(\mathbf{X}')$ is the maximum (over all actions) of the Q value of the state next reached when action a is taken from state $\mathbf{X}$, and β is the fraction of the way toward which the new Q value, $Q(\mathbf{X}, a)$, is adjusted to equal $r + \gamma V(\mathbf{X}')$.

Watkins and Dayan [Watkins & Dayan, 1992] prove that, under certain conditions, the Q values computed by this learning procedure converge to optimal ones (that is, to ones on which an optimal policy can be based).

We define $n^i(\mathbf{X}, a)$ as the index (episode number) of the i-th time that action a is tried in state $\mathbf{X}$. Then, we have:

Theorem 11.1 (Watkins and Dayan) *For Markov problems with states* $\{\mathbf{X}\}$ *and actions* $\{a\}$*, and given bounded rewards* $|r_n| \leq R$*, learning rates* $0 \leq c_n < 1$*, and*

$$\sum_{i=0}^{\infty} c_{n^i(\mathbf{X},a)} = \infty, \quad \sum_{i=0}^{\infty} \left[c_{n^i(\mathbf{X},a)}\right]^2 < \infty$$

for all $\mathbf{X}$ *and* a*, then*
$Q_n(\mathbf{X},a) \to Q_n^*(\mathbf{X},a)$ *as* $n \to \infty$*, for all* $\mathbf{X}$ *and* a*, with probability 1, where* $Q_n^*(\mathbf{X},a)$ *corresponds to the* Q *values of an optimal policy.*

Again, we quote from [Watkins & Dayan, 1992, page 281]:

> "The most important condition implicit in the convergence theorem ... is that the sequence of episodes that forms the basis of learning must include an infinite number of episodes for each starting state and action. This may be considered a strong condition on the way states and actions are selected—however, under the stochastic conditions of the theorem, no method could be guaranteed to find an optimal policy under weaker conditions. Note, however, that the episodes need not form a continuous sequence—that is the $\mathbf{X}'$ of one episode need not be the $\mathbf{X}$ of the next episode."

The relationships among Q learning, dynamic programming, and control are very well described in [Barto, Bradtke, & Singh, 1994]. Q learning is best thought of as a stochastic approximation method for calculating the Q values. Although the definition of the optimal Q values for any state depends recursively on expected values of the Q values for subsequent states (and on the expected values of rewards), no expected values are explicitly computed by the procedure. Instead, these values are approximated by iterative sampling using the actual stochastic mechanism that produces successor states.

11.5 Discussion, Limitations, and Extensions of Q-Learning

11.5.1 An Illustrative Example

The Q-learning procedure requires that we maintain a table of $Q(\mathbf{X},a)$ values for all state-action pairs. In the grid world that we described earlier, such a table would not be excessively large. We might start with random entries in the table; a portion of such an intial table might be as follows:

$\mathbf{X}$	a	$Q(\mathbf{X}, a)$	$r(\mathbf{X}, a)$
(2,3)	w	7	0
(2,3)	n	4	0
(2,3)	e	3	0
(2,3)	s	6	0
(1,3)	w	4	-1
(1,3)	n	5	0
(1,3)	e	2	0
(1,3)	s	4	0

Suppose the robot is in cell (2,3). The maximum Q value occurs for $a = w$, so the robot moves west to cell (1,3)—receiving no immediate reward. The maximum Q value in cell (1,3) is 5, and the learning mechanism attempts to make the value of $Q((2,3), w)$ closer to the discounted value of 5 plus the immediate reward (which was 0 in this case). With a learning rate parameter $c = 0.5$ and $\gamma = 0.9$, the Q value of $Q((2,3), w)$ is adjusted from 7 to 5.75. No other changes are made to the table at this episode. The reader might try this learning procedure on the grid world with a simple computer program. Notice that an optimal policy might not be discovered if some cells are not visited nor some actions not tried frequently enough.

The learning problem faced by the agent is to associate specific actions with specific input patterns. Q learning gradually *reinforces* those actions that contribute to positive rewards by increasing the associated Q values. Typically, as in this example, rewards occur somewhat after the actions that lead to them—hence the phrase *delayed-reinforcement* learning. One can imagine that better and better approximations to the optimal Q values gradually propagate back from states producing rewards toward all of the other states that the agent frequently visits. With random Q values to begin, the agent's actions amount to a random walk through its space of states. Only when this random walk happens to stumble into rewarding states does Q learning begin to produce Q values that are useful, and, even then, the Q values have to work their way outward from these rewarding states. The general problem of associating rewards with state-action pairs is called the *temporal credit assignment problem*—how should credit for a reward be apportioned to the actions leading up to it? Q learning is, to date, the most successful technique for temporal credit assignment, although a related method, called the *bucket brigade algorithm*, has been proposed by [Holland, 1986].

Learning problems similar to that faced by the agent in our grid world

have been thoroughly studied by Sutton who has proposed an architecture, called DYNA, for solving them [Sutton, 1990]. DYNA combines reinforcement learning with planning. Sutton characterizes planning as learning in a simulated world that models the world that the agent inhabits. The agent's model of the world is obtained by Q learning in its actual world, and planning is accomplished by Q learning in its model of the world.

We should note that the learning problem faced by our grid-world robot could be modified to have several places in the grid that give positive rewards. This possibility presents an interesting way to generalize the classical notion of a "goal" in AI planning systems—even in those that do no learning. Instead of representing a goal as a condition to be achieved, we represent a "goal structure" as a set of rewards to be given for achieving various conditions. Then, the generalized "goal" becomes maximizing discounted future reward instead of simply achieving some particular condition. This generalization can be made to encompass so-called goals of *maintenance* and goals of *avoidance*. The example presented above included avoiding bumping into the grid-world boundary. A goal of maintenance, of a particular state, could be expressed in terms of a reward that was earned whenever the agent was in that state and performed an action that transitioned back to that state in one step.

11.5.2 Using Random Actions

When the next pattern presentation in a sequence of patterns is the one caused by the agent's own action in response to the last pattern, we have what is called an *on-line* learning method. In Watkins and Dayan's terminology, in on-line learning the episodes form a continous sequence. As already mentioned, the convergence theorem for Q learning does not require on-line learning; indeed, special precautions must be taken to ensure that on-line learning meets the conditions of the theorem. If on-line learning discovers some good paths to rewards, the agent may fixate on these and never discover a policy that leads to a possibly greater long-term reward. In reinforcement learning phraseology, this problem is referred to as the problem of *exploitation* (of already learned behavior) versus *exploration* (of possibly better behavior).

One way to force exploration is to perform occasional random actions (instead of that single action prescribed by the current Q values). For example, in the grid-world problem, one could imagine selecting an action randomly according to a probability distribution over the actions ($n, e, s,$ and w). This distribution, in turn, could depend on the Q values. For example, we might first find that action prescribed by the Q values and

then choose that action with probability 1/2, choose the two orthogonal actions with probability 3/16 each, and choose the opposite action with probability 1/8. This policy might be modified by "simulated annealing" which would gradually increase the probability of the action prescribed by the Q values more and more as time goes on. This strategy would favor exploration at the beginning of learning and exploitation later.

Other methods, also, have been proposed for dealing with exploration, including making unvisited states intrinsically rewarding and using an "interval estimate," which is related to the uncertainty in the estimate of a state's value [Kaelbling, 1993].

11.5.3 Generalizing Over Inputs

For large problems it would be impractical to maintain a table like that used in our grid-world example. Various researchers have suggested mechanisms for computing Q values, given pattern inputs and actions. One method that suggests itself is to use a neural network. For example, consider the simple linear machine shown in Fig. 11.4.

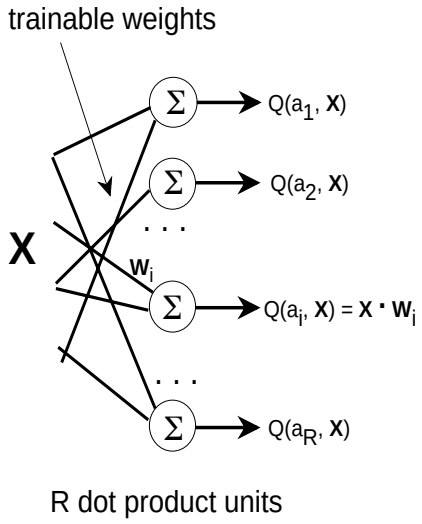


Figure 11.4: A Net that Computes Q Values

Such a neural net could be used by an agent that has R actions to select from. The Q values (as a function of the input pattern $\mathbf{X}$ and the action a_i) are computed as dot products of weight vectors (one for each action) and the input vector. Weight adjustments are made according to a TD(0) procedure to bring the Q value for the action last selected closer to the sum of the immediate reward (if any) and the (discounted) maximum Q value for the next input pattern.

If the optimum Q values for the problem (whatever they might be) are more complex than those that can be computed by a linear machine, a layered neural network might be used. Sigmoid units in the final layer would compute Q values in the range 0 to 1. The TD(0) method for updating Q values would then have to be combined with a multi-layer weight-changing rule, such as backpropagation.

Networks of this sort are able to aggregate different input vectors into regions for which the same action should be performed. This kind of aggregation is an example of what has been called *structural credit assignment*. Combining TD(λ) and backpropagation is a method for dealing with both the temporal and the structural credit assignment problems.

Interesting examples of delayed-reinforcement training of simulated and actual robots requiring structural credit assignment have been reported by [Lin, 1992, Mahadevan & Connell, 1992].

11.5.4 Partially Observable States

So far, we have identified the input vector, $\mathbf{X}$, with the actual state of the environment. When the input vector results from an agent's perceptual apparatus (as we assume it does), there is no reason to suppose that it uniquely identifies the environmental state. Because of inevitable perceptual limitations, several different environmental states might give rise to the same input vector. This phenomenon has been referred to as *perceptual aliasing*. With perceptual aliasing, we can no longer guarantee that Q learning will result in even useful action policies, let alone optimal ones. Several researchers have attempted to deal with this problem using a variety of methods including attempting to model "hidden" states by using internal memory [Lin, 1993]. That is, if some aspect of the environment cannot be sensed currently, perhaps it was sensed once and can be remembered by the agent. When such is the case, we no longer have a Markov problem; that is, the next $\mathbf{X}$ vector, given any action, may depend on a sequence of previous ones rather than just the immediately preceding one. It might be possible to reinstate a Markov framework (over the $\mathbf{X}$'s) if $\mathbf{X}$

includes not only current sensory precepts but information from the agent's memory.

11.5.5 Scaling Problems

Several difficulties have so far prohibited wide application of reinforcement learning to large problems. (The TD-gammon program, mentioned in the last chapter, is probably unique in terms of success on a high-dimensional problem.) We have already touched on some difficulties; these and others are summarized below with references to attempts to overcome them.

a. Exploration versus exploitation.

- use random actions
- favor states not visited recently
- separate the learning phase from the use phase
- employ a teacher to guide exploration

b. Slow time to convergence

- combine learning with prior knowledge; use estimates of Q values (rather than random values) initially
- use a hierarchy of actions; learn primitive actions first and freeze the useful sequences into macros and then learn how to use the macros
- employ a teacher; use graded "lessons"—starting near the rewards and then backing away, and use examples of good behavior [Lin, 1992]
- use more efficient computations; *e.g.* do several updates per episode [Moore & Atkeson, 1993]

c. Large state spaces

- use hand-coded features
- use neural networks
- use nearest-neighbor methods [Moore, 1990]

d. Temporal discounting problems. Using small γ can make the learner too greedy for present rewards and indifferent to the future; but using large γ slows down learning.

- use a learning method based on average rewards [Schwartz, 1993]

e. No "transfer" of learning . What is learned depends on the reward structure; if the rewards change, learning has to start over.

- Separate the learning into two parts: learn an "action model" which predicts how actions change states (and is constant over all problems), and then learn the "values" of states by reinforcement learning for each different set of rewards. Sometimes the reinforcement learning part can be replaced by a "planner" that uses the action model to produce plans to achieve goals.

Also see other articles in the special issue on reinforcement learning: *Machine Learning*, 8, May, 1992.

11.6 Bibliographical and Historical Remarks

To be added.

Chapter 12

Explanation-Based Learning

12.1 Deductive Learning

In the learning methods studied so far, typically the training set does not exhaust the version space. Using logical terminology, we could say that the classifier's output does not logically follow from the training set. In this sense, these methods are *inductive*. In logic, a *deductive* system is one whose conclusions logically follow from a set of input facts, if the system is sound.[1]

To contrast inductive with deductive systems in a logical setting, suppose we have a set of facts (the training set) that includes the following formulas:

$$\{Round(Obj1), Round(Obj2), Round(Obj3), Round(Obj4),$$

$$Ball(Obj1), Ball(Obj2), Ball(Obj3), Ball(Obj4)\}$$

A learning system that forms the conclusion $(\forall x)[Ball(x) \supset Round(x)]$ is inductive. This conclusion may be useful (if there are no facts of the form $Ball(\sigma) \wedge \neg Round(\sigma)$), but it does not logically follow from the facts. On the other hand, if we had the facts $Green(Obj5)$ and $Green(Obj5) \supset$

[1] Logical reasoning systems that are not sound, for example those using non-monotonic reasoning, themselves might produce inductive conclusions that do not logically follow from the input facts.

$Round(Obj5)$, then we could logically conclude $Round(Obj5)$. Making this conclusion and saving it is an instance of deductive learning—a topic we study in this chapter.

Suppose that some logical proposition, ϕ, logically follows from some set of facts, Δ. Under what circumstances might we say that the process of deducing ϕ from Δ results in our *learning* ϕ? In a sense, we implicitly knew ϕ all along, since it was inherent in knowing Δ. Yet, ϕ might not be obvious given Δ, and the deduction process to establish ϕ might have been arduous. Rather than have to deduce ϕ again, we might want to save it, perhaps along with its deduction, in case it is needed later. Shouldn't that process count as learning? Dietterich [Dietterich, 1990] has called this type of learning *speed-up* learning.

Strictly speaking, speed-up learning does not result in a system being able to make decisions that, in principle, could not have been made before the learning took place. Speed-up learning simply makes it possible to make those decisions more efficiently. But, in practice, this type of learning might make possible certain decisions that might otherwise have been infeasible.

To take an extreme case, a chess player can be said to learn chess even though optimal play is inherent in the rules of chess. On the surface, there seems to be no real difference between the experience-based hypotheses that a chess player makes about what constitutes good play and the kind of learning we have been studying so far.

As another example, suppose we are given some theorems about geometry and are asked to prove that the sum of the angles of a right triangle is 180 degrees. Let us further suppose that the proof we constructed did not depend on the given triangle being a right triangle; in that case we can learn a more general fact. The learning technique that we are going to study next is related to this example. It is called *explanation-based learning (EBL)*. EBL can be thought of as a process in which *implicit* knowledge is converted into *explicit* knowledge.

In EBL, we *specialize* parts of a *domain theory* to *explain* a particular *example*, then we *generalize* the explanation to produce another element of the domain theory that will be useful on similar examples. This process is illustrated in Fig. 12.1.

12.2 Domain Theories

Two types of information were present in the inductive methods we have studied: the information inherent in the training samples and the information about the domain that is implied by the "bias" (for example, the

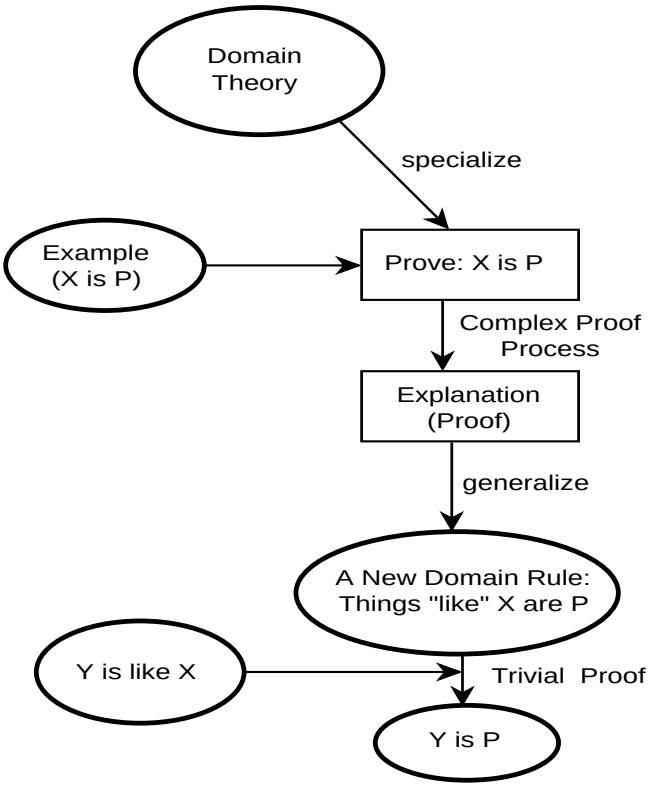


Figure 12.1: The EBL Process

hypothesis set from which we choose functions). The learning methods are successful only if the hypothesis set is appropriate for the problem. Typically, the smaller the hypothesis set (that is, the more a priori information we have about the function being sought), the less dependent we are on information being supplied by a training set (that is, fewer samples). A priori information about a problem can be expressed in several ways. The methods we have studied so far restrict the hypotheses in a rather direct way. A less direct method involves making assertions in a logical language about the property we are trying to learn. A set of such assertions is usually called a "domain theory."

Suppose, for example, that we wanted to classify people according to whether or not they were good credit risks. We might represent a person

by a set of properties (income, marital status, type of employment, *etc.*), assemble such data about people who are known to be good and bad credit risks and train a classifier to make decisions. Or, we might go to a loan officer of a bank, ask him or her what sorts of things s/he looks for in making a decision about a loan, encode this knowledge into a set of rules for an expert system, and then use the expert system to make decisions. The knowledge used by the loan officer might have originated as a set of "policies" (the domain theory), but perhaps the application of these policies were specialized and made more efficient through experience with the special cases of loans made in his or her district.

12.3 An Example

To make our discussion more concrete, let's consider the following fanciful example. We want to find a way to classify robots as "robust" or not. The attributes that we use to represent a robot might include some that are relevant to this decision and some that are not.

Suppose we have a domain theory of logical sentences that taken together, help to define whether or not a robot can be classified as robust. (The same domain theory may be useful for several other purposes also, but among other things, it describes the concept "robust.")

In this example, let's suppose that our domain theory includes the sentences:

$$Fixes(u, u) \supset Robust(u)$$

(An individual that can fix itself is robust.)

$$Sees(x, y) \wedge Habile(x) \supset Fixes(x, y)$$

(A habile individual that can see another entity can fix that entity.)

$$Robot(w) \supset Sees(w, w)$$

(All robots can see themselves.)

$$R2D2(x) \supset Habile(x)$$

(R2D2-class individuals are habile.)

$$C3PO(x) \supset Habile(x)$$

(C3PO-class individuals are habile.)

...

(By convention, variables are assumed to be universally quantified.) We could use theorem-proving methods operating on this domain theory to conclude whether certain robots are robust. These methods might be computationally quite expensive because extensive search may have to be performed to derive a conclusion. But after having found a proof for some particular robot, we might be able to derive some new sentence whose use allows a much faster conclusion.

We next show how such a new rule might be derived in this example. Suppose we are given a number of facts about Num5, such as:

$$Robot(Num5)$$

$$R2D2(Num5)$$

$$Age(Num5, 5)$$

$$Manufacturer(Num5, GR)$$

...

We are also told that $Robust(Num5)$ is true, but we nevertheless attempt to find a proof of that assertion using these facts about Num5 and the domain theory. The facts about Num5 correspond to the features that we might use to represent Num5. In this example, not all of them are relevant to a decision about $Robust(Num5)$. The relevant ones are those used or needed in proving $Robust(Num5)$ using the domain theory. The proof tree in Fig. 12.2 is one that a typical theorem-proving system might produce.

In the language of EBL, this proof is an *explanation* for the fact $Robust(Num5)$. We see from this explanation that the only facts about Num5 that were used were $Robot(Num5)$ and $R2D2(Num5)$. In fact, we could construct the following rule from this explanation:

$$Robot(Num5) \wedge R2D2(Num5) \supset Robust(Num5)$$

The explanation has allowed us to prune some attributes about Num5 that are irrelevant (at least for deciding $Robust(Num5)$). This type of pruning is the first sense in which an explanation is used to generalize the classification problem. ([DeJong & Mooney, 1986] call this aspect of explanation-based

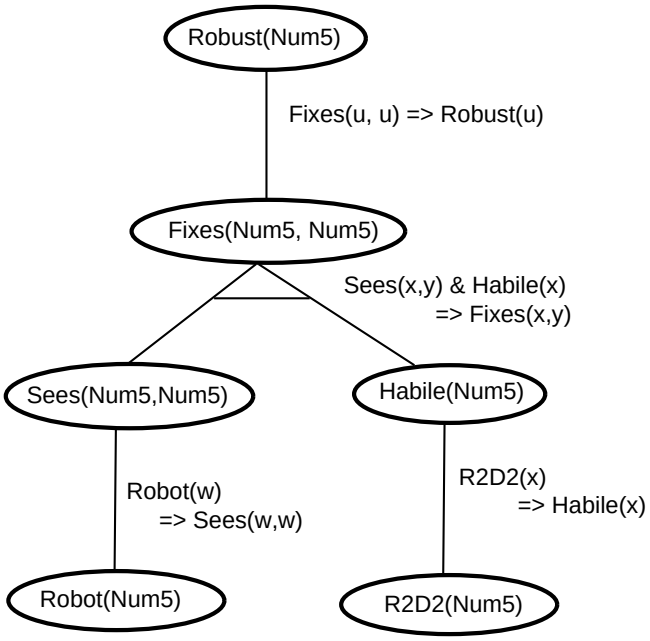


Figure 12.2: A Proof Tree

learning *feature elimination.*) But the rule we extracted from the explanation applies only to Num5. There might be little value in learning that rule since it is so specific. Can it be generalized so that it can be applied to other individuals as well?

Examination of the proof shows that the same proof structure, using the same sentences from the domain theory, could be used independently of whether we are talking about Num5 or some other individual. We can generalize the proof by a process that replaces constants in the tip nodes of the proof tree with variables and works upward—using unification to constrain the values of variables as needed to obtain a proof.

In this example, we replace $Robot(Num5)$ by $Robot(r)$ and $R2D2(Num5)$ by $R2D2(s)$ and redo the proof—using the explanation proof as a template. Note that we use different values for the two different occurrences of $Num5$ at the tip nodes. Doing so sometimes results in more general, but nevertheless valid rules. We now apply the rules used in the proof in the forward direction, keeping track of the substitutions imposed by the most general

unifiers used in the proof. (Note that we always substitute terms that are already in the tree for variables in rules.) This process results in the generalized proof tree shown in Fig. 12.3. Note that the occurrence of $Sees(r, r)$ as a node in the tree forces the unification of x with y in the domain rule, $Sees(x, y) \wedge Habile(y) \supset Fixes(x, y)$. The substitutions are then applied to the variables in the tip nodes and the root node to yield the general rule: $Robot(r) \wedge R2D2(r) \supset Robust(r)$.

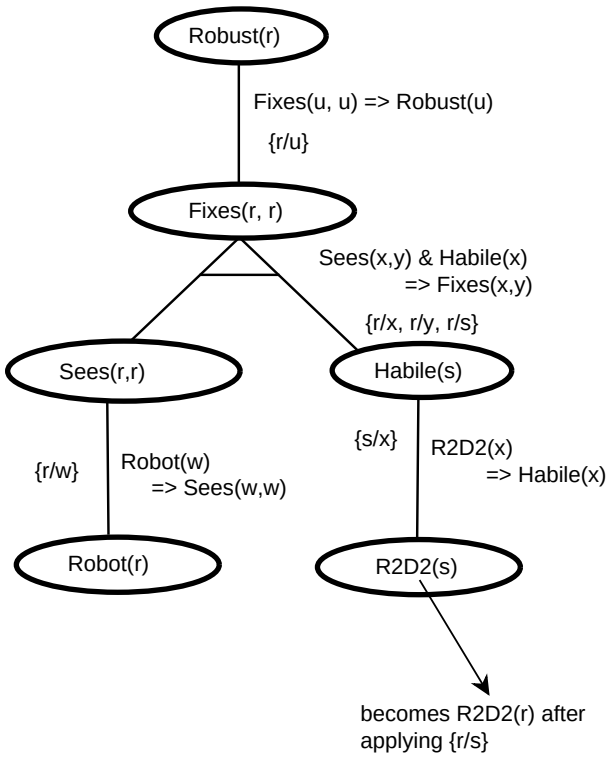


Figure 12.3: A Generalized Proof Tree

This rule is the end result of EBL for this example. The process by which $Num5$ in this example was generalized to a variable is what [DeJong & Mooney, 1986] call *identity elimination* (the precise identity of Num5 turned out to be irrelevant). (The generalization process described in this example is based on that of [DeJong & Mooney, 1986] and differs from that of [Mitchell, *et al.*, 1986]. It is also similar to that used

in [Fikes, *et al.*, 1972].) Clearly, under certain assumptions, this general rule is more easily used to conclude *Robust* about an individual than the original proof process was.

It is important to note that we could have derived the general rule from the domain theory without using the example. (In the literature, doing so is called *static analysis* [Etzioni, 1991].) In fact, the example told us nothing new other than what it told us about Num5. The sole role of the example in this instance of EBL was to provide a template for a proof to help guide the generalization process. Basing the generalization process on examples helps to insure that we learn rules matched to the distribution of problems that occur.

There are a number of qualifications and elaborations about EBL that need to be mentioned.

12.4 Evaluable Predicates

The domain theory includes a number of predicates other than the one occuring in the formula we are trying to prove and other than those that might customarily be used to describe an individual. One might note, for example, that if we used $Habile(Num5)$ to describe Num5, the proof would have been shorter. Why didn't we? The situation is analogous to that of using a data base augmented by logical rules. In the latter application, the formulas in the actual data base are "extensional," and those in the logical rules are "intensional." This usage reflects the fact that the predicates in the data base part are defined by their extension—we *explicitly* list all the tuples sastisfying a relation. The logical rules serve to connect the data base predicates with higher level abstractions that are described (if not defined) by the rules. We typically cannot look up the truth values of formulas containing these intensional predicates; they have to be derived using the rules and the database.

The EBL process assumes something similar. The domain theory is useful for connecting formulas that we might want to prove with those whose truth values can be "looked up" or otherwise evaluated. In the EBL literature, such formulas satisfy what is called the *operationality criterion.* Perhaps another analogy might be to neural networks. The evaluable predicates correspond to the components of the input pattern vector; the predicates in the domain theory correspond to the hidden units. Finding the new rule corresponds to finding a simpler expression for the formula to be proved in terms only of the evaluable predicates.

12.5 More General Proofs

Examining the domain theory for our example reveals that an alternative rule might have been: $Robot(u) \wedge C3PO(u) \supset Robust(u)$. Such a rule might have resulted if we were given $\{C3PO(Num6), Robot(Num6), \ldots\}$ and proved $Robust(Num6)$. After considering these two examples (Num5 and Num6), the question arises, do we want to generalize the two rules to something like: $Robot(u) \wedge [C3PO(u) \vee R2D2(u)] \supset Robust(u)$? Doing so is an example of what [DeJong & Mooney, 1986] call *structural generalization* (via *disjunctive augmentation*).

Adding disjunctions for every alternative proof can soon become cumbersome and destroy any efficiency advantage of EBL. In our example, the efficiency might be retrieved if there were another evaluable predicate, say, $Bionic(u)$ such that the domain theory also contained $R2D2(x) \supset Bionic(x)$ and $C3PO(x) \supset Bionic(x)$. After seeing a number of similar examples, we might be willing to *induce* the formula $Bionic(u) \supset [C3PO(u) \vee R2D2(u)]$ in which case the rule with the disjunction could be replaced with $Robot(u) \wedge Bionic(u) \supset Robust(u)$.

12.6 Utility of EBL

It is well known in theorem proving that the complexity of finding a proof depends both on the number of formulas in the domain theory and on the depth of the shortest proof. Adding a new rule decreases the depth of the shortest proof but it also increases the number of formulas in the domain theory. In realistic applications, the added rules will be relevant for some tasks and not for others. Thus, it is unclear whether the overall utility of the new rules will turn out to be positive. EBL methods have been applied in several settings, usually with positive utility. (See [Minton, 1990] for an analysis).

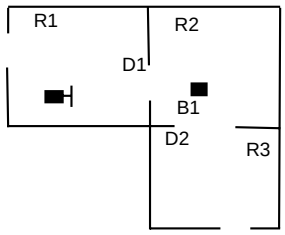
12.7 Applications

There have been several applications of EBL methods. We mention two here, namely the formation of macro-operators in automatic plan generation and learning how to control search.

12.7.1 Macro-Operators in Planning

In automatic planning systems, efficiency can sometimes be enhanced by chaining together a sequence of operators into *macro-operators*. We show an example of a process for creating macro-operators based on techniques explored by [Fikes, *et al.*, 1972].

Referring to Fig. 12.4, consider the problem of finding a plan for a robot in room $R1$ to fetch a box, $B1$, by going to an adjacent room, $R2$, and pushing it back to $R1$. The goal for the robot is $INROOM(B1, R1)$, and the facts that are true in the initial state are listed in the figure.



Initial State:

INROOM(ROBOT, R1)
INROOM(B1,R2)
CONNECTS(D1,R1,R2)
CONNECTS(D1,R2,R1)

. . .

Figure 12.4: Initial State of a Robot Problem

We will construct the plan from a set of STRIPS operators that include:

GOTHRU$(d, r1, r2)$

Preconditions: $INROOM(ROBOT, r1), CONNECTS(d, r1, r2)$

Delete list: $INROOM(ROBOT, r1)$

Add list: $INROOM(ROBOT, r2)$

PUSHTHRU$(b, d, r1, r2)$

Preconditions: $INROOM(ROBOT, r1), CONNECTS(d, r1, r2), INROOM(b, r1)$

Delete list: $INROOM(ROBOT, r1), INROOM(b, r1)$

Add list: $INROOM(ROBOT, r2), INROOM(b, r2)$

A backward-reasoning STRIPS system might produce the plan shown in Fig. 12.5. We show there the main goal and the subgoals along a solution path. (The conditions in each subgoal that are true in the initial state are shown underlined.) The preconditions for this plan, true in the initial state, are:

$$INROOM(ROBOT, R1)$$

$$CONNECTS(D1, R1, R2)$$

$$CONNECTS(D1, R2, R1)$$

$$INROOM(B1, R2)$$

Saving this specific plan, valid only for the specific constants it mentions, would not be as useful as would be saving a more general one. We first generalize these preconditions by substituting variables for constants. We then follow the structure of the specific plan to produce the generalized plan shown in Fig. 12.6 that achieves $INROOM(b1, r4)$. Note that the generalized plan does not require pushing the box back to the place where the robot started. The preconditions for the generalized plan are:

$$INROOM(ROBOT, r1)$$

$$CONNECTS(d1, r1, r2)$$

$$CONNECTS(d2, r2, r4)$$

$$INROOM(b, r4)$$

Another related technique that chains together sequences of operators to form more general ones is the *chunking* mechanism in Soar [Laird, *et al.*, 1986].

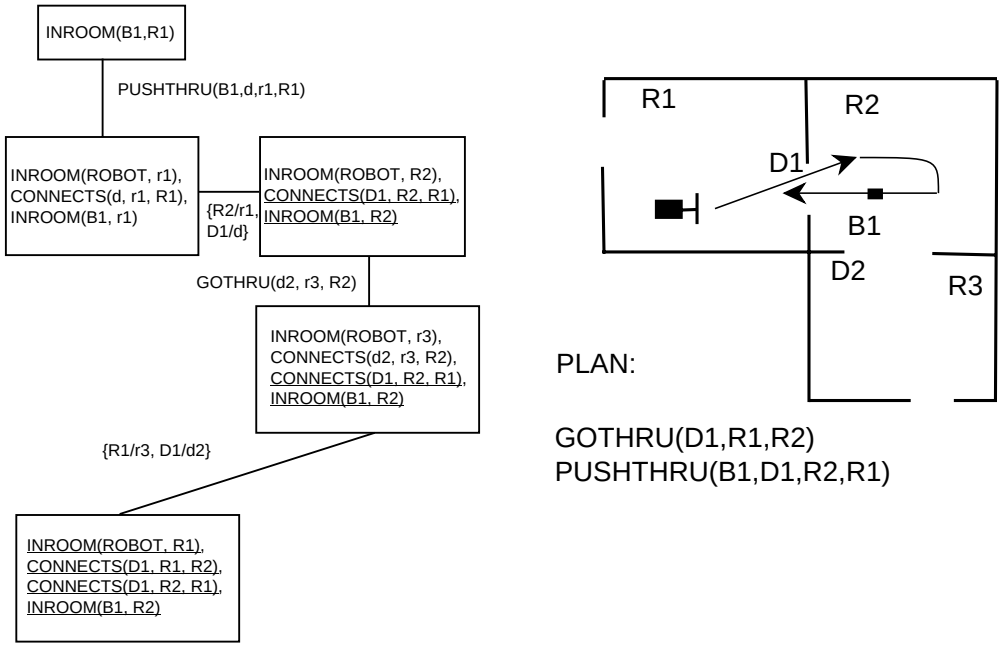


Figure 12.5: A Plan for the Robot Problem

12.7.2 Learning Search Control Knowledge

Besides their use in creating macro-operators, EBL methods can be used to improve the efficiency of planning in another way also. In his system called PRODIGY, Minton proposed using EBL to learn effective ways to control search [Minton, 1988]. PRODIGY is a STRIPS-like system that solves planning problems in the blocks-world, in a simple mobile robot world, and in job-shop scheduling. PRODIGY has a domain theory involving both the domain of the problem and a simple (meta) theory about planning. Its meta theory includes statements about whether a control choice about a subgoal to work on, an operator to apply, *etc.* either *succeeds* or *fails*. After producing a plan, it analyzes its successful and its unsuccessful choices and attempts to explain them in terms of its domain theory. Using an EBL-like process, it is able to produce useful control rules such as:

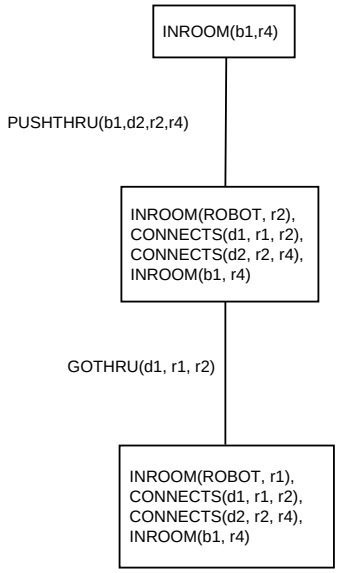


Figure 12.6: A Generalized Plan

```
IF (AND (CURRENT-NODE node)
        (CANDIDATE-GOAL node (ON  x y))
        (CANDIDATE-GOAL node (ON y z)))
THEN (PREFER GOAL (ON y z) TO (ON  x y))
```

PRODIGY keeps statistics on how often these learned rules are used, their savings (in time to find plans), and their cost of application. It saves only the rules whose utility, thus measured, is judged to be high. Minton [Minton, 1990] has shown that there is an overall advantage of using these rules (as against not having any rules and as against hand-coded search control rules).

12.8 Bibliographical and Historical Remarks

To be added.

Bibliography

[Acorn & Walden, 1992] Acorn, T., and Walden, S., "SMART: Support Management Automated Reasoning Technology for COMPAQ Customer Service," *Proc. Fourth Annual Conf. on Innovative Applications of Artificial Intelligence*, Menlo Park, CA: AAAI Press, 1992.

[Aha, 1991] Aha, D., Kibler, D., and Albert, M., "Instance-Based Learning Algorithms," *Machine Learning*, 6, 37-66, 1991.

[Anderson & Bower, 1973] Anderson, J. R., and Bower, G. H., *Human Associative Memory*, Hillsdale, NJ: Erlbaum, 1973.

[Anderson, 1958] Anderson, T. W., *An Introduction to Multivariate Statistical Analysis*, New York: John Wiley, 1958.

[Barto, Bradtke, & Singh, 1994] Barto, A., Bradtke, S., and Singh, S., "Learning to Act Using Real-Time Dynamic Programming," to appear in *Artificial Intelligence*, 1994.

[Baum & Haussler, 1989] Baum, E, and Haussler, D., "What Size Net Gives Valid Generalization?" *Neural Computation*, 1, pp. 151-160, 1989.

[Baum, 1994] Baum, E., "When Are k-Nearest Neighbor and Backpropagation Accurate for Feasible-Sized Sets of Examples?" in Hanson, S., Drastal, G., and Rivest, R., (eds.), *Computational Learning Theory and Natural Learning Systems, Volume 1: Constraints and Prospects*, pp. 415-442, Cambridge, MA: MIT Press, 1994.

[Bellman, 1957] Bellman, R. E., *Dynamic Programming*, Princeton: Princeton University Press, 1957.

[Blumer, *et al.*, 1987] Blumer, A., *et al.*, "Occam's Razor," *Info. Process. Lett., vol 24*, pp. 377-80, 1987.

[Blumer, *et al.*, 1990] Blumer, A., *et al.*, "Learnability and the Vapnik-Chervonenkis Dimension," *JACM*, 1990.

[Bollinger & Duffie, 1988] Bollinger, J., and Duffie, N., *Computer Control of Machines and Processes*, Reading, MA: Addison-Wesley, 1988.

[Brain, *et al.*, 1962] Brain, A. E., *et al.*, "Graphical Data Processing Research Study and Experimental Investigation," Report No. 8 (pp. 9-13) and No. 9 (pp. 3-10), Contract DA 36-039 SC-78343, SRI International, Menlo Park, CA, June 1962 and September 1962.

[Breiman, *et al.*, 1984] Breiman, L., Friedman, J., Olshen, R., and Stone, C., *Classification and Regression Trees*, Monterey, CA: Wadsworth, 1984.

[Brent, 1990] Brent, R. P., "Fast Training Algorithms for Multi-Layer Neural Nets," Numerical Analysis Project Manuscript NA-90-03, Computer Science Department, Stanford University, Stanford, CA 94305, March 1990.

[Bryson & Ho 1969] Bryson, A., and Ho, Y.-C., *Applied Optimal Control*, New York: Blaisdell.

[Buchanan & Wilkins, 1993] Buchanan, B. and Wilkins, D., (eds.), *Readings in Knowledge Acquisition and Learning*, San Francisco: Morgan Kaufmann, 1993.

[Carbonell, 1983] Carbonell, J., "Learning by Analogy," in *Machine Learning: An Artificial Intelligence Approach*, Michalski, R., Carbonell, J., and Mitchell, T., (eds.), San Francisco: Morgan Kaufmann, 1983.

[Cheeseman, *et al.*, 1988] Cheeseman, P., *et al.*, "AutoClass: A Bayesian Classification System," *Proc. Fifth Intl. Workshop on Machine Learning*, Morgan Kaufmann, San Mateo, CA, 1988. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, Morgan Kaufmann, San Francisco, pp. 296-306, 1990.

[Cover & Hart, 1967] Cover, T., and Hart, P., "Nearest Neighbor Pattern Classification," *IEEE Trans. on Information Theory*, 13, 21-27, 1967.

[Cover, 1965] Cover, T., "Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition," *IEEE Trans. Elec. Comp.*, EC-14, 326-334, June, 1965.

[Dasarathy, 1991] Dasarathy, B. V., *Nearest Neighbor Pattern Classification Techniques*, IEEE Computer Society Press, 1991.

[Dayan & Sejnowski, 1994] Dayan, P., and Sejnowski, T., "$TD(\lambda)$ Converges with Probability 1," *Machine Learning*, 14, pp. 295-301, 1994.

[Dayan, 1992] Dayan, P., "The Convergence of TD(λ) for General λ," *Machine Learning*, 8, 341-362, 1992.

[DeJong & Mooney, 1986] DeJong, G., and Mooney, R., "Explanation-Based Learning: An Alternative View," *Machine Learning*, 1:145-176, 1986. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp 452-467.

[Dietterich & Bakiri, 1991] Dietterich, T. G., and Bakiri, G., "Error-Correcting Output Codes: A General Method for Improving Multiclass Inductive Learning Programs," *Proc. Ninth Nat. Conf. on A.I.*, pp. 572-577, AAAI-91, MIT Press, 1991.

[Dietterich, *et al.*, 1990] Dietterich, T., Hild, H., and Bakiri, G., "A Comparative Study of ID3 and Backpropagation for English Text-to-Speech Mapping," *Proc. Seventh Intl. Conf. Mach. Learning*, Porter, B. and Mooney, R. (eds.), pp. 24-31, San Francisco: Morgan Kaufmann, 1990.

[Dietterich, 1990] Dietterich, T., "Machine Learning," *Annu. Rev. Comput. Sci.*, 4:255-306, Palo Alto: Annual Reviews Inc., 1990.

[Duda & Fossum, 1966] Duda, R. O., and Fossum, H., "Pattern Classification by Iteratively Determined Linear and Piecewise Linear Discriminant Functions," *IEEE Trans. on Elect. Computers*, vol. EC-15, pp. 220-232, April, 1966.

[Duda & Hart, 1973] Duda, R. O., and Hart, P.E., *Pattern Classification and Scene Analysis*, New York: Wiley, 1973.

[Duda, 1966] Duda, R. O., "Training a Linear Machine on Mislabeled Patterns," SRI Tech. Report prepared for ONR under Contract 3438(00), SRI International, Menlo Park, CA, April 1966.

[Efron, 1982] Efron, B., *The Jackknife, the Bootstrap and Other Resampling Plans*, Philadelphia: SIAM, 1982.

[Ehrenfeucht, *et al.*, 1988] Ehrenfeucht, A., *et al.*, "A General Lower Bound on the Number of Examples Needed for Learning," in *Proc. 1988 Workshop on Computational Learning Theory*, pp. 110-120, San Francisco: Morgan Kaufmann, 1988.

[Etzioni, 1991] Etzioni, O., "STATIC: A Problem-Space Compiler for PRODIGY," *Proc. of Ninth National Conf. on Artificial Intelligence*, pp. 533-540, Menlo Park: AAAI Press, 1991.

[Etzioni, 1993] Etzioni, O., "A Structural Theory of Explanation-Based Learning," *Artificial Intelligence*, 60:1, pp. 93-139, March, 1993.

[Evans & Fisher, 1992] Evans, B., and Fisher, D., *Process Delay Analyses Using Decision-Tree Induction*, Tech. Report CS92-06, Department of Computer Science, Vanderbilt University, TN, 1992.

[Fahlman & Lebiere, 1990] Fahlman, S., and Lebiere, C., "The Cascade-Correlation Learning Architecture," in Touretzky, D., (ed.), *Advances in Neural Information Processing Systems, 2*, pp. 524-532, San Francisco: Morgan Kaufmann, 1990.

[Fayyad, *et al.*, 1993] Fayyad, U. M., Weir, N., and Djorgovski, S., "SKICAT: A Machine Learning System for Automated Cataloging of Large Scale Sky Surveys," in *Proc. Tenth Intl. Conf. on Machine Learning*, pp. 112-119, San Francisco: Morgan Kaufmann, 1993. (For a longer version of this paper see: Fayyad, U. Djorgovski, G., and Weir, N., "Automating the Analysis and Cataloging of Sky Surveys," in Fayyad, U., *et al.*(eds.), *Advances in Knowledge Discovery and Data Mining*, Chapter 19, pp. 471ff., Cambridge: The MIT Press, March, 1996.)

[Feigenbaum, 1961] Feigenbaum, E. A., "The Simulation of Verbal Learning Behavior," *Proceedings of the Western Joint Computer Conference*, 19:121-132, 1961.

[Fikes, *et al.*, 1972] Fikes, R., Hart, P., and Nilsson, N., "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, pp 251-288, 1972. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp 468-486.

[Fisher, 1987] Fisher, D., "Knowledge Acquisition via Incremental Conceptual Clustering," *Machine Learning*, 2:139-172, 1987. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 267–283.

[Friedman, *et al.*, 1977] Friedman, J. H., Bentley, J. L., and Finkel, R. A., "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Trans. on Math. Software*, 3(3):209-226, September 1977.

[Fu, 1994] Fu, L., *Neural Networks in Artificial Intelligence*, New York: McGraw-Hill, 1994.

[Gallant, 1986] Gallant, S. I., "Optimal Linear Discriminants," in *Eighth International Conf. on Pattern Recognition*, pp. 849-852, New York: IEEE, 1986.

[Genesereth & Nilsson, 1987] Genesereth, M., and Nilsson, N., *Logical Foundations of Artificial Intelligence*, San Francisco: Morgan Kaufmann, 1987.

[Gluck & Rumelhart, 1989] Gluck, M. and Rumelhart, D., *Neuroscience and Connectionist Theory*, The Developments in Connectionist Theory, Hillsdale, NJ: Erlbaum Associates, 1989.

[Hammerstrom, 1993] Hammerstrom, D., "Neural Networks at Work," *IEEE Spectrum*, pp. 26-32, June 1993.

[Haussler, 1988] Haussler, D., "Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework," *Artificial Intelligence*, 36:177-221, 1988. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 96-107.

[Haussler, 1990] Haussler, D., "Probably Approximately Correct Learning," *Proc. Eighth Nat. Conf. on AI*, pp. 1101-1108. Cambridge, MA: MIT Press, 1990.

[Hebb, 1949] Hebb, D. O., *The Organization of Behaviour*, New York: John Wiley, 1949.

[Hertz, Krogh, & Palmer, 1991] Hertz, J., Krogh, A, and Palmer, R., *Introduction to the Theory of Neural Computation*, Lecture Notes, vol. 1, Santa Fe Inst. Studies in the Sciences of Complexity, New York: Addison-Wesley, 1991.

[Hirsh, 1994] Hirsh, H., "Generalizing Version Spaces," *Machine Learning*, 17, 5-45, 1994.

[Holland, 1975] Holland, J., *Adaptation in Natural and Artificial Systems*, Ann Arbor: The University of Michigan Press, 1975. (Second edition printed in 1992 by MIT Press, Cambridge, MA.)

[Holland, 1986] Holland, J. H., "Escaping Brittleness; The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems." In Michalski, R., Carbonell, J., and Mitchell, T. (eds.) , *Machine Learning: An Artificial Intelligence Approach, Volume 2*, chapter 20, San Francisco: Morgan Kaufmann, 1986.

[Hunt, Marin, & Stone, 1966] Hunt, E., Marin, J., and Stone, P., *Experiments in Induction*, New York: Academic Press, 1966.

[Jabbour, K., *et al.*, 1987] Jabbour, K., *et al.*, "ALFA: Automated Load Forecasting Assistant," *Proc. of the IEEE Pwer Engineering Society Summer Meeting*, San Francisco, CA, 1987.

[John, 1995] John, G., "Robust Linear Discriminant Trees," *Proc. of the Conf. on Artificial Intelligence and Statistics*, Ft. Lauderdale, FL, January, 1995.

[Kaelbling, 1993] Kaelbling, L. P., *Learning in Embedded Systems*, Cambridge, MA: MIT Press, 1993.

[Kohavi, 1994] Kohavi, R., "Bottom-Up Induction of Oblivious Read-Once Decision Graphs," *Proc. of European Conference on Machine Learning (ECML-94)*, 1994.

[Kolodner, 1993] Kolodner, J., *Case-Based Reasoning*, San Francisco: Morgan Kaufmann, 1993.

[Koza, 1992] Koza, J., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press, 1992.

[Koza, 1994] Koza, J., *Genetic Programming II: Automatic Discovery of Reusable Programs*, Cambridge, MA: MIT Press, 1994.

[Laird, *et al.*, 1986] Laird, J., Rosenbloom, P., and Newell, A., "Chunking in Soar: The Anatomy of a General Learning Mechanism," *Machine Learning*, 1, pp. 11-46, 1986. Reprinted in Buchanan, B. and Wilkins, D., (eds.), *Readings in Knowledge Acquisition and Learning*, pp. 518-535, Morgan Kaufmann, San Francisco, CA, 1993.

[Langley, 1992] Langley, P., "Areas of Application for Machine Learning," *Proc. of Fifth Int'l. Symp. on Knowledge Engineering*, Sevilla, 1992.

[Langley, 1996] Langley, P., *Elements of Machine Learning*, San Francisco: Morgan Kaufmann, 1996.

[Lavrač & Džeroski, 1994] Lavrač, N., and Džeroski, S., *Inductive Logic Programming*, Chichester, England: Ellis Horwood, 1994.

[Lin, 1992] Lin, L., "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning, and Teaching," *Machine Learning*, 8, 293-321, 1992.

[Lin, 1993] Lin, L., "Scaling Up Reinforcement Learning for Robot Control," *Proc. Tenth Intl. Conf. on Machine Learning*, pp. 182-189, San Francisco: Morgan Kaufmann, 1993.

[Littlestone, 1988] Littlestone, N., "Learning Quickly When Irrelevant Attributes Abound: A New Linear-Threshold Algorithm," *Machine Learning* 2: 285-318, 1988.

[Maass & Turán, 1994] Maass, W., and Turán, G., "How Fast Can a Threshold Gate Learn?," in Hanson, S., Drastal, G., and Rivest, R., (eds.), *Computational Learning Theory and Natural Learning Systems, Volume 1: Constraints and Prospects*, pp. 381-414, Cambridge, MA: MIT Press, 1994.

[Mahadevan & Connell, 1992] Mahadevan, S., and Connell, J., "Automatic Programming of Behavior-Based Robots Using Reinforcement Learning," *Artificial Intelligence*, 55, pp. 311-365, 1992.

[Marchand & Golea, 1993] Marchand, M., and Golea, M., "On Learning Simple Neural Concepts: From Halfspace Intersections to Neural Decision Lists," *Network*, 4:67-85, 1993.

[McCulloch & Pitts, 1943] McCulloch, W. S., and Pitts, W. H., "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, Vol. 5, pp. 115-133, Chicago: University of Chicago Press, 1943.

[Michie, 1992] Michie, D., "Some Directions in Machine Intelligence," unpublished manuscript, The Turing Institute, Glasgow, Scotland, 1992.

[Minton, 1988] Minton, S., *Learning Search Control Knowledge: An Explanation-Based Approach*, Kluwer Academic Publishers, Boston, MA, 1988.

[Minton, 1990] Minton, S., "Quantitative Results Concerning the Utility of Explanation-Based Learning," *Artificial Intelligence*, 42, pp. 363-392, 1990. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 573-587.

[Mitchell, *et al.*, 1986] Mitchell, T., *et al.*, "Explanation-Based Generalization: A Unifying View," *Machine Learning*, 1:1, 1986. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 435-451.

[Mitchell, 1982] Mitchell, T., "Generalization as Search," *Artificial Intelligence*, 18:203-226, 1982. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 96–107.

[Moore & Atkeson, 1993] Moore, A., and Atkeson, C., "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time," *Machine Learning*, 13, pp. 103-130, 1993.

[Moore, *et al.*, 1994] Moore, A. W., Hill, D. J., and Johnson, M. P., "An Empirical Investigation of Brute Force to Choose Features, Smoothers, and Function Approximators," in Hanson, S., Judd, S., and Petsche, T., (eds.), *Computational Learning Theory and Natural Learning Systems*, Vol. 3, Cambridge: MIT Press, 1994.

[Moore, 1990] Moore, A., *Efficient Memory-based Learning for Robot Control*, PhD. Thesis; Technical Report No. 209, Computer Laboratory, University of Cambridge, October, 1990.

[Moore, 1992] Moore, A., "Fast, Robust Adaptive Control by Learning Only Forward Models," in Moody, J., Hanson, S., and Lippman, R., (eds.), *Advances in Neural Information Processing Systems 4*, San Francisco: Morgan Kaufmann, 1992.

[Mueller & Page, 1988] Mueller, R. and Page, R., *Symbolic Computing with Lisp and Prolog*, New York: John Wiley & Sons, 1988.

[Muggleton, 1991] Muggleton, S., "Inductive Logic Programming," *New Generation Computing*, 8, pp. 295-318, 1991.

[Muggleton, 1992] Muggleton, S., *Inductive Logic Programming*, London: Academic Press, 1992.

[Muroga, 1971] Muroga, S., *Threshold Logic and its Applications*, New York: Wiley, 1971.

[Natarjan, 1991] Natarajan, B., *Machine Learning: A Theoretical Approach*, San Francisco: Morgan Kaufmann, 1991.

[Nilsson, 1965] Nilsson, N. J., "Theoretical and Experimental Investigations in Trainable Pattern-Classifying Systems," Tech. Report No. RADC-TR-65-257, Final Report on Contract AF30(602)-3448, Rome Air Development Center (Now Rome Laboratories), Griffiss Air Force Base, New York, September, 1965.

[Nilsson, 1990] Nilsson, N. J., *The Mathematical Foundations of Learning Machines*, San Francisco: Morgan Kaufmann, 1990. (This book is a reprint of *Learning Machines: Foundations of Trainable Pattern-Classifying Systems*, New York: McGraw-Hill, 1965.)

[Oliver, Dowe, & Wallace, 1992] Oliver, J., Dowe, D., and Wallace, C., "Inferring Decision Graphs using the Minimum Message Length Principle," *Proc. 1992 Australian Artificial Intelligence Conference*, 1992.

[Pagallo & Haussler, 1990] Pagallo, G. and Haussler, D., "Boolean Feature Discovery in Empirical Learning," *Machine Learning*, vol.5, no.1, pp. 71-99, March 1990.

[Pazzani & Kibler, 1992] Pazzani, M., and Kibler, D., "The Utility of Knowledge in Inductive Learning," *Machine Learning*, 9, 57-94, 1992.

[Peterson, 1961] Peterson, W., *Error Correcting Codes*, New York: John Wiley, 1961.

[Pomerleau, 1991] Pomerleau, D., "Rapidly Adapting Artificial Neural Networks for Autonomous Navigation," in Lippmann, P., *et al.* (eds.), *Advances in Neural Information Processing Systems, 3*, pp. 429-435, San Francisco: Morgan Kaufmann, 1991.

[Pomerleau, 1993] Pomerleau, D, *Neural Network Perception for Mobile Robot Guidance*, Boston: Kluwer Academic Publishers, 1993.

[Quinlan & Rivest, 1989] Quinlan, J. Ross, and Rivest, Ron, "Inferring Decision Trees Using the Minimum Description Length Principle," *Information and Computation*, 80:227–248, March, 1989.

[Quinlan, 1986] Quinlan, J. Ross, "Induction of Decision Trees," *Machine Learning*, 1:81–106, 1986. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 57–69.

[Quinlan, 1987] Quinlan, J. R., "Generating Production Rules from Decision Trees," In *IJCAI-87: Proceedings of the Tenth Intl. Joint Conf. on Artificial Intelligence*, pp. 304-7, San Francisco: Morgan-Kaufmann, 1987.

[Quinlan, 1990] Quinlan, J. R., "Learning Logical Definitions from Relations," *Machine Learning*, 5, 239-266, 1990.

[Quinlan, 1993] Quinlan, J. Ross, *C4.5: Programs for Machine Learning*, San Francisco: Morgan Kaufmann, 1993.

[Quinlan, 1994] Quinlan, J. R., "Comparing Connectionist and Symbolic Learning Methods," in Hanson, S., Drastal, G., and Rivest, R., (eds.), *Computational Learning Theory and Natural Learning Systems, Volume 1: Constraints and Prospects*, pp. 445-456,, Cambridge, MA: MIT Press, 1994.

[Ridgway, 1962] Ridgway, W. C., *An Adaptive Logic System with Generalizing Properties*, PhD thesis, Tech. Rep. 1556-1, Stanford Electronics Labs., Stanford, CA, April 1962.

[Rissanen, 1978] Rissanen, J., "Modeling by Shortest Data Description," *Automatica*, 14:465-471, 1978.

[Rivest, 1987] Rivest, R. L., "Learning Decision Lists," *Machine Learning*, 2, 229-246, 1987.

[Rosenblatt, 1958] Rosenblatt, F., *Principles of Neurodynamics*, Washington: Spartan Books, 1961.

[Ross, 1983] Ross, S., *Introduction to Stochastic Dynamic Programming*, New York: Academic Press, 1983.

[Rumelhart, Hinton, & Williams, 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning Internal Representations by Error Propagation," In Rumelhart, D. E., and McClelland, J. L., (eds.) *Parallel Distributed Processing*, Vol 1, 318–362, 1986.

[Russell & Norvig 1995] Russell, S., and Norvig, P., *Artificial Intelligence: A Modern Approach*, Englewood Cliffs, NJ: Prentice Hall, 1995.

[Samuel, 1959] Samuel, A., "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, 3:211-229, July 1959.

[Schwartz, 1993] Schwartz, A., "A Reinforcement Learning Method for Maximizing Undiscounted Rewards," *Proc. Tenth Intl. Conf. on Machine Learning*, pp. 298-305, San Francisco: Morgan Kaufmann, 1993.

[Sejnowski, Koch, & Churchland, 1988] Sejnowski, T., Koch, C., and Churchland, P., "Computational Neuroscience," *Science*, **241**: 1299-1306, 1988.

[Shavlik, Mooney, & Towell, 1991] Shavlik, J., Mooney, R., and Towell, G., "Symbolic and Neural Learning Algorithms: An Experimental Comparison," *Machine Learning*, 6, pp. 111-143, 1991.

[Shavlik & Dietterich, 1990] Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990.

[Sutton & Barto, 1987] Sutton, R. S., and Barto, A. G., "A Temporal-Difference Model of Classical Conditioning," in *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Erlbaum, 1987.

[Sutton, 1988] Sutton, R. S., "Learning to Predict by the Methods of Temporal Differences," *Machine Learning* 3: 9-44, 1988.

[Sutton, 1990] Sutton, R., "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," *Proc. of the Seventh Intl. Conf. on Machine Learning*, pp. 216-224, San Francisco: Morgan Kaufmann, 1990.

[Taylor, Michie, & Spiegalhalter, 1994] Taylor, C., Michie, D., and Spiegalhalter, D., *Machine Learning, Neural and Statistical Classification*, Paramount Publishing International.

[Tesauro, 1992] Tesauro, G., "Practical Issues in Temporal Difference Learning," *Machine Learning*, 8, nos. 3/4, pp. 257-277, 1992.

[Towell & Shavlik, 1992] Towell G., and Shavlik, J., "Interpretation of Artificial Neural Networks: Mapping Knowledge-Based Neural Networks into Rules," in Moody, J., Hanson, S., and Lippmann, R., (eds.), *Advances in Neural Information Processing Systems, 4*, pp. 977-984, San Francisco: Morgan Kaufmann, 1992.

[Towell, Shavlik, & Noordweier, 1990] Towell, G., Shavlik, J., and Noordweier, M., "Refinement of Approximate Domain Theories by Knowledge-Based Artificial Neural Networks," *Proc. Eighth Natl., Conf. on Artificial Intelligence*, pp. 861-866, 1990.

[Unger, 1989] Unger, S., *The Essence of Logic Circuits*, Englewood Cliffs, NJ: Prentice-Hall, 1989.

[Utgoff, 1989] Utgoff, P., "Incremental Induction of Decision Trees," *Machine Learning*, 4:161–186, Nov., 1989.

[Valiant, 1984] Valiant, L., "A Theory of the Learnable," *Communications of the ACM, Vol. 27*, pp. 1134-1142, 1984.

[Vapnik & Chervonenkis, 1971] Vapnik, V., and Chervonenkis, A., "On the Uniform Convergence of Relative Frequencies, *Theory of Probability and its Applications, Vol. 16*, No. 2, pp. 264-280, 1971.

[Various Editors, 1989-1994] *Advances in Neural Information Processing Systems*, vols 1 through 6, San Francisco: Morgan Kaufmann, 1989 -1994.

[Watkins & Dayan, 1992] Watkins, C. J. C. H., and Dayan, P., "Technical Note: Q-Learning," *Machine Learning*, 8, 279-292, 1992.

[Watkins, 1989] Watkins, C. J. C. H., *Learning From Delayed Rewards*, PhD Thesis, University of Cambridge, England, 1989.

[Weiss & Kulikowski, 1991] Weiss, S., and Kulikowski, C., *Computer Systems that Learn*, San Francisco: Morgan Kaufmann, 1991.

[Werbos, 1974] Werbos, P., *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. Thesis, Harvard University, 1974.

[Widrow & Lehr, 1990] Widrow, B., and Lehr, M. A., "30 Years of Adaptive Neural Networks: Perceptron, Madaline and Backpropagation," *Proc. IEEE*, vol. 78, no. 9, pp. 1415-1442, September, 1990.

[Widrow & Stearns, 1985] Widrow, B., and Stearns, S., *Adaptive Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall.

[Widrow, 1962] Widrow, B., "Generalization and Storage in Networks of Adaline Neurons," in Yovits, Jacobi, and Goldstein (eds.), *Self-organizing Systems—1962*, pp. 435-461, Washington, DC: Spartan Books, 1962.

[Winder, 1961] Winder, R., "Single Stage Threshold Logic," *Proc. of the AIEE Symp. on Switching Circuits and Logical Design*, Conf. paper CP-60-1261, pp. 321-332, 1961.

[Winder, 1962] Winder, R., *Threshold Logic*, PhD Dissertation, Princeton University, Princeton, NJ, 1962.

[Wnek, *et al.*, 1990] Wnek, J., *et al.*, "Comparing Learning Paradigms via Diagrammatic Visualization," in *Proc. Fifth Intl. Symp. on Methodologies for Intelligent Systems*, pp. 428-437, 1990. (Also Tech. Report MLI90-2, University of Illinois at Urbana-Champaign.)